ASN1C

ASN.1 Compiler
Version 6.2
C# User's Manual

Objective Systems, Inc. version 6.2 — May 2009

The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of this agreement.

Copyright Notice

Copyright ©1997-2009 Objective Systems, Inc. All rights reserved.

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information

Comments, suggestions, and inquiries regarding ASN1C may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

Overview of ASN1C for C# 1
Using the Compiler 3
ASN1C C# Command Line Options 3
Using the GUI Wizard to Run ASN1C 7
C Sharp Code Generation Options 12
Compilation 16
Compiler Configuration File 16
Compiler Error Reporting 19
Generated C# Source Code Overview 21
Namespace Specification 22
Class Declaration 22
Tag Constant 23
Public Member Variables 23
Constructors 23
Decode Method 24
Encode Method 24
Other Methods 25
Inner Classes 25
ASN.1 Type to C# Class Mappings 27
BOOLEAN 27
INTEGER 27
BIT STRING 28
OCTET STRING 29
Character String Types 30
ENUMERATED 31
NULL 32
OBJECT IDENTIFIER 32
RELATIVE-OID 33
REAL 33
REAL (Base 10) 33
SEQUENCE 34
Creation of Temporary Types 35
OPTIONAL keyword 36
DEFAULT keyword 36
Extension Elements 37
XSD <xsd:all> Type Mapping 37
SET 37
SEQUENCE OF 38
Generation of Temporary Types for SEQUENCE OF Elements 38
SEQUENCE OF Type Elements in Other Constructed Types 39
SET OF 40
CHOICE 40
Creation of Temporary Types 41
Populating Generated Choice Structures for Encoding 41
Accessing the Choice Element Value after Decoding 42
XSD <xsd:union> Type Mapping 42
Open Type 42
<xsd:any> Handling 43
External Type 43
EmbeddedPDV Type 44
Parameterized Types 44

Value Specifications 46
INTEGER Value Specification 46
BOOLEAN Value Specification 47
Binary String Value Specification 47
Hexadecimal String Value Specification 47
Character String Value Specification 47
Object Identifier Value Specification 48
ENUMERATED Value Specification 48
REAL Value Specification 48
SEQUENCE Value Specification 48
SET Value Specification 49
SEQUENCE OF Value Specification 49
SET OF Value Specification 50
CHOICE Value Specification 50
Generated BER/DER/CER Encode Methods 51
Memory-buffer Based Definite Length Encoders 51
Generated C# Method Format and Calling Parameters 51
Populating Generated Variables for Encoding 52
Procedure for Calling Memory-Buffer Based BER Encode Methods 52
Reuse of C# Encoding Objects 54
Stream-Oriented Indefinite Length Encode Methods 55
Generated C# Method Format and Calling Parameters 55
Procedure for Calling C# BER Stream-Oriented Encode Methods 56
Generated BER/DER/CER Decode Methods 59
Generated C# Method Format and Calling Parameters 59
Procedure for Calling C# BER Decode Methods 59
Reuse of C# Decoding Objects 60
Generated PER Encode Methods 63
Generated C# Method Format and Calling Parameters 63
Procedure for Calling C# PER Encode Methods 64
Reuse of C# Encoding Objects 65
Generated PER Decode Methods 67
Generated C# Method Format and Calling Parameters 67
Procedure for Calling C# PER Decode Methods 67
Reuse of C# Decoding Objects 68
Generated XER / XML Encode Methods 71
Generated C# Method Format and Calling Parameters 71
Procedure for Calling C# XER Encode Methods 72
Generated XER / XML Decode Methods 77
Table Constraint Processing 79
CLASS Specification 79
Data Member Generation 79
Method and Constructor Generation 80
ABSTRACT-SYNTAX class 81
TYPE-IDENTIFIER class 82
Information Object 82
Information Object Set 83
Generated Information Object Table Structure 84
Simple Form Code Generation 85
Table Form Code Generation 86
Additional Code Generated for the -tables Option 86
Populating OpenType Variables for Encoding 88
Decoding Types with Table Constraints 90
Generated Print Methods 91

Generated C# Method Format and Calling Parameters 91
Generated Makefile 93
Event Handler Interface 95
How It Works 95
How to Use It 96
Example 1: A Formatted Print Handler 96
Example 2: An XML Converter Class 98
IMPORT/EXPORT of Types 103
Compact Code Generation 105
ASN1C90 107
ROSE OPERATION and ERROR 107
SNMP OBJECT-TYPE 110

Overview of ASN1C for C#

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) or XML Schema Definitions (XSD) source file into computer language source files that allow typed data to be encoded/decoded. This release of ASN1C includes options to generate code in the following languages: C, C++, C# or Java. This manual discusses the C# code generation capabilities. The following other manuals discuss the other language code generation capabilities:

- *ASN1C C/C++ Compiler User's Manual* : C/C++ code generation
- *ASN1C Java Compiler User's Manual* : Java code generation

Each module or namespace that is encountered in an ASN.1 or XSD source file results in the generation of a series of C# source files. A separate C# file is generated for each production (type or global element) in the source file. Additional files are generated for compiler-generated productions and to hold value specification constants.

There is also a set of classes that form the run-time component of the C# package. These classes provide the primitive component building blocks that are assembled by the compiler to encode/decode complex structures. They also provide support for managing message buffers that hold the encoded message components.

ASN1C works with the version of ASN.1 specified in ITU-T international standards X.680 through X.683. It generates code for encoding/decoding data in accordance with the following encoding rules:

- Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), and Canonical Encoding Rules (CER) as published in the ITU-T X.690 standard.
- Packed Encoding Rules (PER) as published in the ITU-T X.691 standard.
- XML Encoding Rules (XER) as published in the ITU-T X.693 standard.
- XML Schema to ASN.1 translation as published in the ITU-T X.694 standard.

The compiler is capable of parsing all ASN.1 syntax as defined in the standards. It is capable of parsing advanced syntax including Information Object Specifications as defined in the ITU-T X.681 standard as well as Parameterized Types as defined in ITU-T X.683.

Note that XER support does not include support for the EXTENDED-XER syntax. This is accomplished through direct compilation of XSD files. An internal translation of XSD to ASN.1 based on the rules in the X.694 standard is done within the compiler and the resulting ASN.1 syntax is compiled into C# classes.

This release of the compiler contains a special compiler option (-1990) that is backward compatible with deprecated features from the older X.208 and X.209 standards. These include the ANY data type and unnamed fields in SEQUENCE, SET, and CHOICE types. This version can also parse type syntax from common macro definitions such as ROSE and SNMP.

Using the Compiler

The ASN1C compiler distribution contains command-line compiler executables as well as a graphical user interface (GUI) wizard that can aid in the specification of compiler options. Please refer to the ASN1C C/C++ Compiler User's Manual for instructions on how to run the compiler. The remaining sections describe options and configuration items specific to the C# version.

ASN1C C# Command Line Options

The following table shows a summary of the command line options that have meaning when C# code generation is selected:

Option	Argument	Description
-asnstd	x680 x208 mixed	This option instructs the compiler to parse ASN.1 syntax conforming to the specified standard. x680 (the default) refers to modern ASN.1 as specified in the ITU-T X.680-X.690 series of standards. x208 refers to the now deprecated X.208 and X.209 standards. This syntax allowed the ANY construct as well as unnamed fields in SEQUENCE, SET, and CHOICE constructs. This option also allows for parsing and generation of code for ROSE OPERATION and ERROR macros and SNMP OBJECT-TYPE macros. The mixed option is used to specify a source file that contains modules with both X.208 and X.680 based syntax.
-ber	None	This option instructs the compiler to generate functions that implement the Basic Encoding Rules (BER) as specified in the ASN.1 standards.
-c#	None	Generate C# source code.
-cer	None	This option instructs the compiler to generate functions that implement the Canonical Encoding Rules (CER) as specified in the ASN.1 standards.
-compact	None	This option instructs the compiler to generate more compact code at the expense of some constraint and error checking. This is an optimization option that should be used after an application is thoroughly tested.
-compat	<versionNumber>	Generate code compatible with an older version of the compiler. The compiler will attempt to generate code more closely aligned with the given

Option	Argument	Description
		previous release of the compiler. <versionNumber> is specified as x.x (for example, -compat 5.2)
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. A full discussion of the contents of a configuration file is provided in the *Compiler Configuration File* section.
-depends	None	This option instructs the compiler to generate a full set of C# source files that contain only the productions in the main file being compiled and items those productions depend on from IMPORT files.
-der	None	This option instructs the compiler to generate functions that implement the Distinguished Encoding Rules (DER) as specified in the ASN.1 standards.
-dirs	None	This is a C# option that causes a subdirectory to be created to hold each of the generated C# source files for each module in an ASN.1 source file.
-events	None	Generate extra code to invoke user defined event and error handler callback methods (see the *Event Handlers* section).
-genmake	None	This option is used to generate a makefile for compiling generated classes using *nmake* command from within a Visual Studio .NET command prompt window.
-I	<directory>	This option is used to specify a directory that the compiler will search for ASN.1 source files for IMPORT items. Multiple –I qualifiers can be used to specify multiple directories to search.
-lax	None	This option instructs the compiler to not generate code to check constraints. When used in conjunction with the *–compact* option (described next), it produces the smallest code base for a given ASN.1 specification.
-list	None	Generate listing. This will dump the source code to the standard output device as it is parsed. This can be useful for finding parse errors.

Option	Argument	Description
-namespace	<namespaceName>	This is a C# option that allows the entire C# namespace to be changed. Instead of the module name, the full name specified using this option will be used. This option cannot be used in conjunction with –nspfx option.
-noencode	None	This option suppresses the generation of encode functions.
-nodecode	None	This option suppresses the generation of decode functions.
-noIndefLen	None	This option instructs the compiler to omit indefinite length tests in generated decode functions. These tests result in the generation of a large amount of code. If you know that your application only uses definite length encoding, this option can result in a much smaller code base size.
-noOpenExt	None	This option instructs the compiler to not add an open extension element in constructs that contain extensibility markers. The purpose of the element is to collect any unknown items in a message. If an application does not care about these unknown items, it can use this option to reduce the size of the generated code.
-nspfx	<prefixName>	This is a C# option for adding a prefix in front of the assigned C# namespace name. By default, the C# namespace name is set to the module name. If the namespace is embedded within a hierarchy, this option can be used to set the prefix that must be added to allow C# to find the class definition.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-pdu	<typeName>	Designate given type name to be a Protocol Definition Unit (PDU) type. By default, PDU types are determined to be types that are not referenced by any other types within a module. This option allows that behavior to be overridden. The * wildcard character may be specified for <typeName> to indicate that all productions within an ASN.1 module should be treated as PDU types.

Option	Argument	Description
-per	None	This option instructs the compiler to generate functions that implement the Packed Encoding Rules (PER) as specified in the ASN.1 standards.
-genPrint -print	None	This option specifies that print methods should be generated. Print
-shortnames	None	This option is used to change the name generated by compiler for embedded types in constructed types. This option is required to handle the limit on the size of filenames in certain situations. With this option, the generated code filenames would be shorter than without this option.
-stream	None	This option instructs the compiler to generate stream-based encoders/decoders instead of memory buffer based. This makes it possible to encode directly to or decode directly from a source or sink such as a file or socket. In the case of BER, it will also cause forward encoders to be generated, which will use indefinite lengths for all constructed elements in a message.
-tables	None	This option is used to generate additional code for the handling of table constraints as defined in the X.682 standard.
-uniquenames	None	This option instructs the compiler to automatically generate unique names to resolve name collisions in the generated code. Name collisions can occur, for example, if two modules are being compiled that contain a production with the same name. A un ique name is generated by prepending the module name to one of the productions to form a name of the form <module>_<name>. Note that name collisions can also be manually resolved by using the typePrefix, enumPrefix, and valuePrefix configuration items (see the Compiler Configuration File section for more details).
-warnings	None	Output information on compiler generated warnings.
-xer	None	This option instructs the compiler to generate functions that implement the

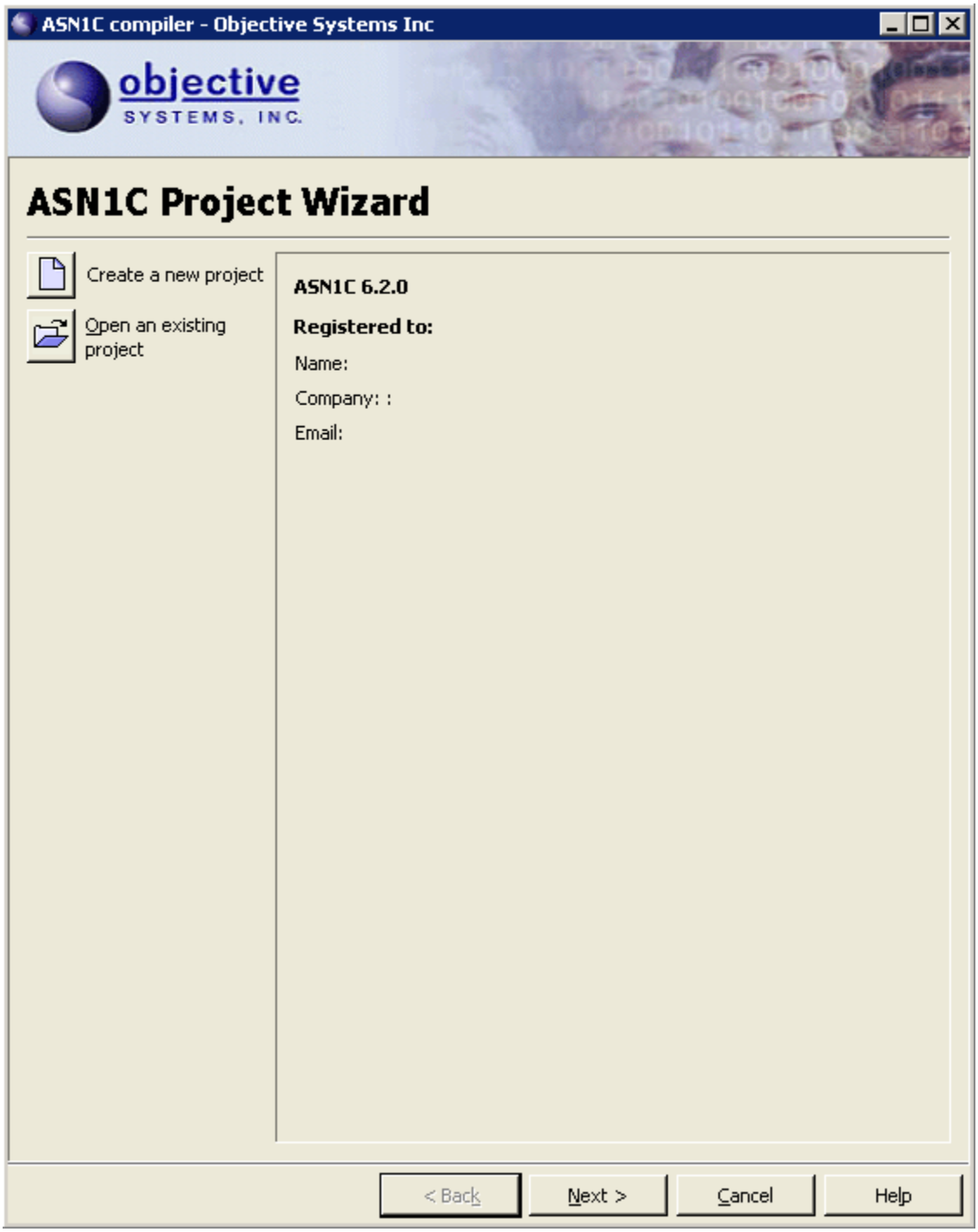
Option	Argument	Description
		XML Encoding Rules (XER) as specified in the ASN.1 standards.
-xml	None	This option instructs the compiler to generate functions that implement the XML Encoding Rules (XML) as specified in the World-Wide Consortium (W3C). Related XML Schema can be produced by using the -xsd command line option.

Using the GUI Wizard to Run ASN1C

ASN1C includes a graphical user interface (GUI) wizard that can be used as an alternative to the command-line version. It is a cross-platform GUI and has been ported to Windows and most UNIXes. The GUI makes it possible to specify ASN.1 files and configuration files via file navigation windows, to set command line options by checking boxes, and to get online help on specific options.

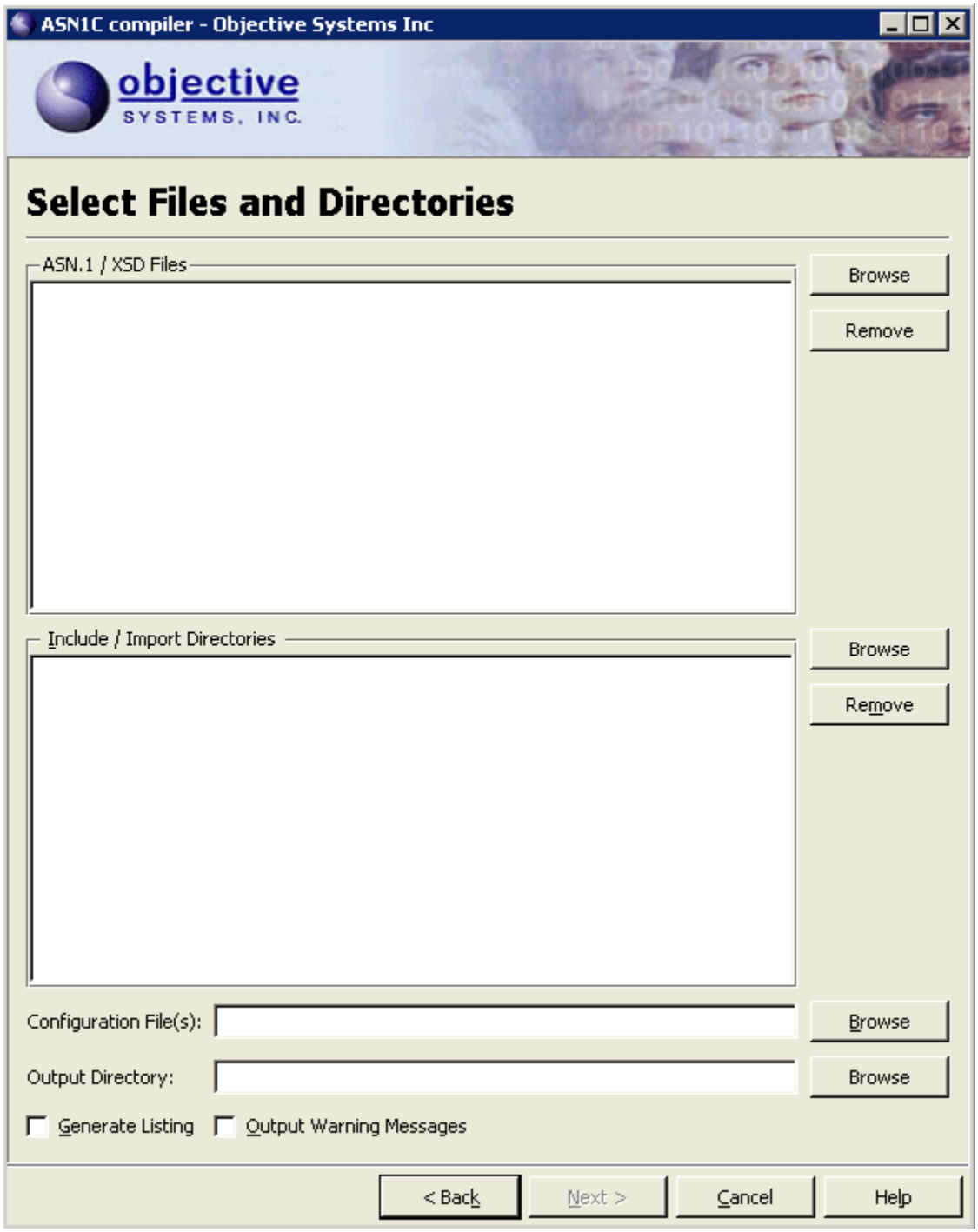
The Windows installation program should have installed an 'ASN1C Compiler' option on your computer desktop and an 'ASN1C' option on the start menu. The wizard can be launched using either of these items. The UNIX version should be installed in `ASN1C_INSTALL_DIR/bin`; no desktop shortcuts are created, so it will be necessary to create one or to run the wizard from the command-line.

The wizard is navigated by means of *Next* and *Back* buttons. Following is the initial window:



The Project Wizard will allow you to save your compilation options and file settings into a project file and retrieve them later. If you wish to make a new project, click the icon next to *Create a New Project*. Previously saved projects may be recalled by clicking the icon next to *Open an Existing Project*.

The status window will display the version of the software you have installed as well as report any errors upon startup that occur, such as a missing license file.



In this window, the ASN.1 file or files to be compiled are selected. This is done by clicking the *Add* button on the right hand side of the top windows pane. A file selection box will appear allowing you to select the ASN.1 or XSD files to be compiled. Files can be removed from the pane by highlighting the entry and clicking the *Remove* button.

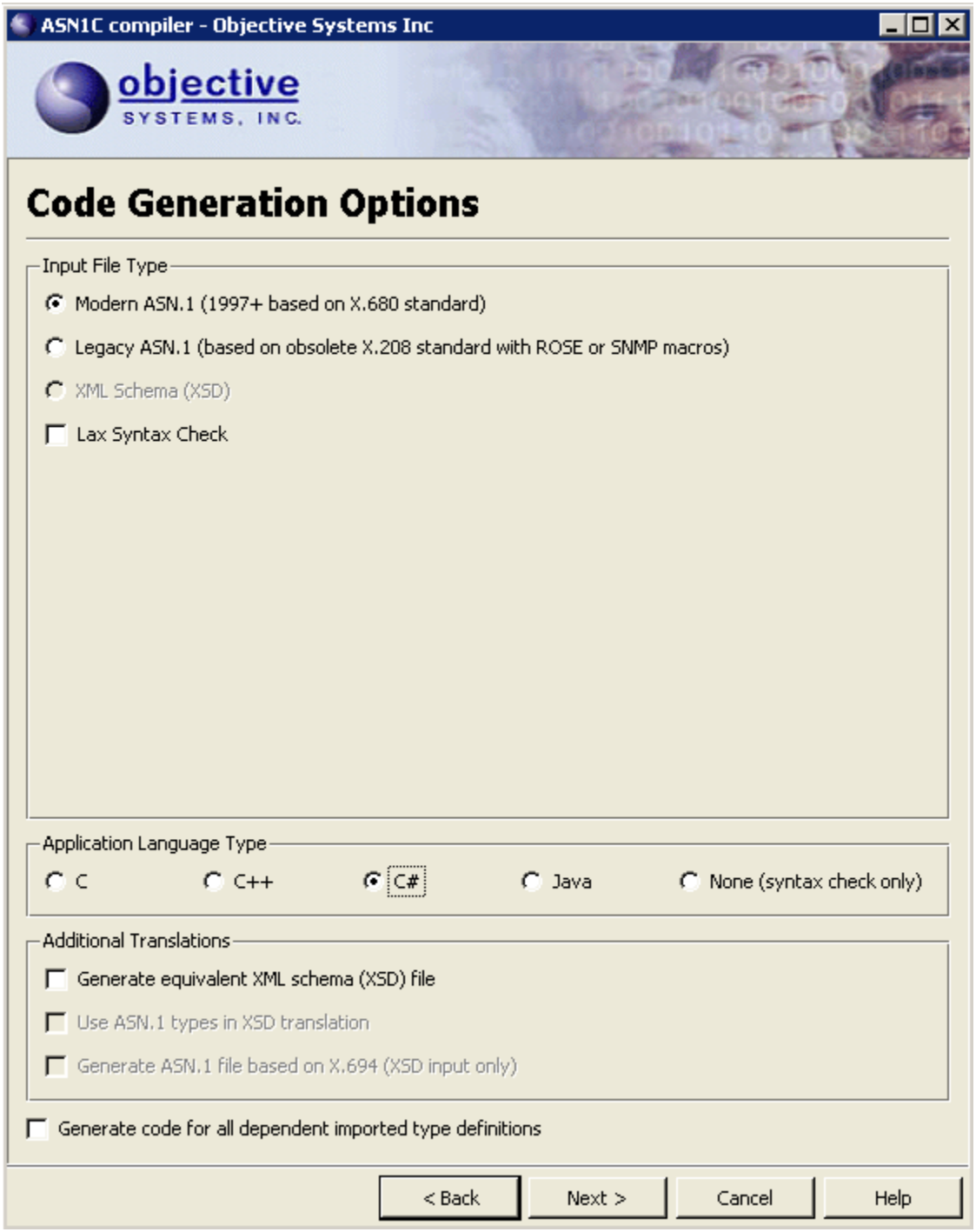
ASN.1 specifications and XML Schema Documents must not be compiled in the same project. Once an ".asn" file has been added, no ".xsd" files may be added.

Include directories are selected in a similar manner in the middle pane. These are directories the compiler will search for import files. By default, the compiler looks for files in the current working directory with the name of the module being imported and extension ".asn" or ".xsd". Additional directories can be searched for these files by adding them here.

User-defined configuration files are specified in the third pane. These allow further control of the compilation process. They are optional and are only needed if the default compilation process is to be altered (for example, if a type prefix is to be added to a generated type name). See the *Compiler Configuration File* section for details on defining these files.

There are also two options to generate extra compilation information. *Check Generate* Listing if you want the compiler to echo the specification as it compiles it. Check *Output Warning Messages* to output potential problems that occurred during compilation.

The next window is as follows:



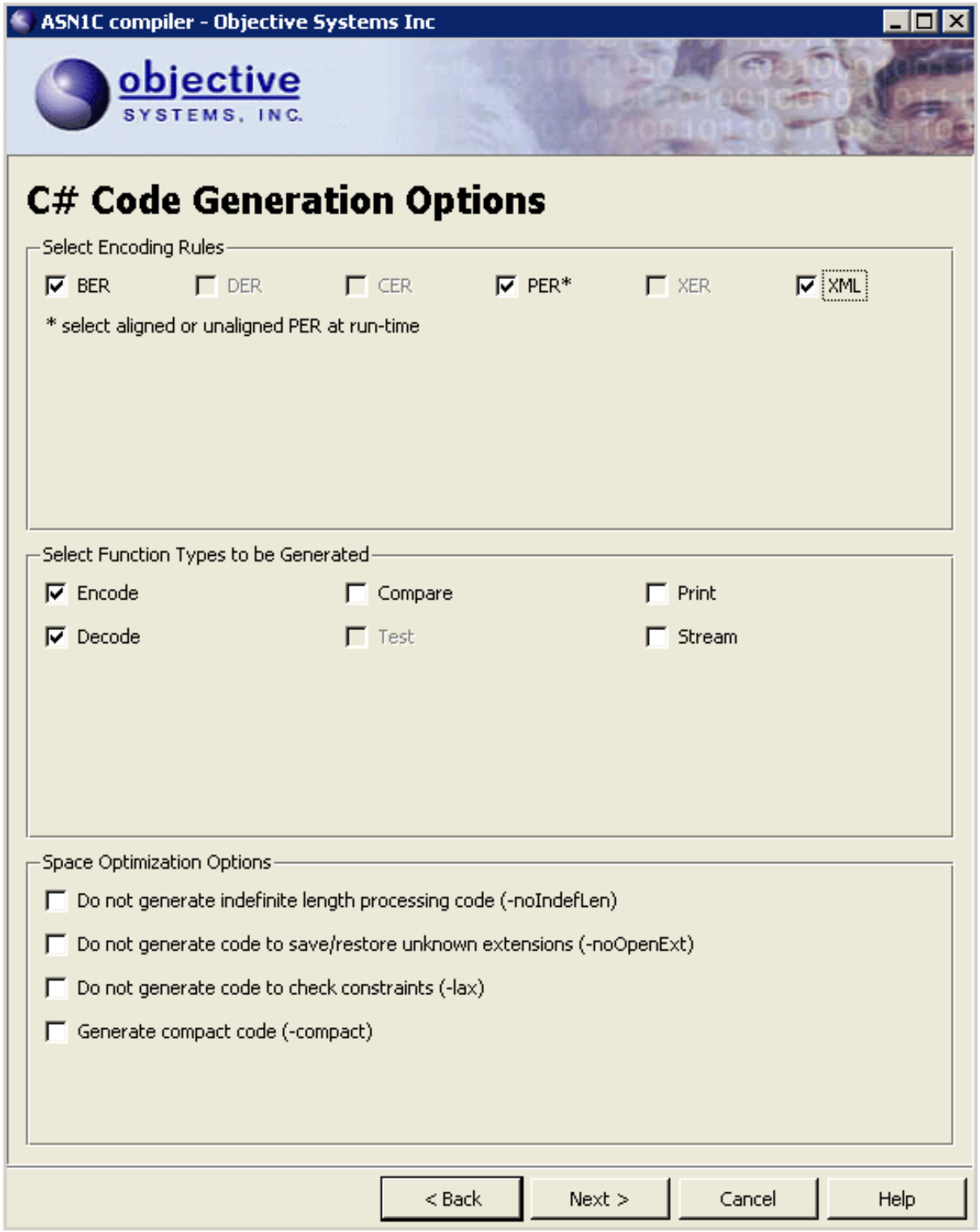
The *Code Generations Options* window permits users to specify the input language type, target application language, and additional translations if necessary. If C# was the only language option purchased, trying to use other languages will result in compiler errors.

Certain options will be inactive (greyed out) depending on the file type selected. For example, if an XSD file is selected, the option *Generate ASN.1 file based on X.694* will be active and the option *Generate equivalent XML schema (XSD) file* will be inactive.

Checking *Generate code for all dependent imported type definitions* will cause the compiler to search and generate code for modules specified in the IMPORTS statement of an ASN.1 specification.

C Sharp Code Generation Options

The following screens detail options that may be used when the target language is C Sharp:



This dialog permits users to modify the code that is generated by the compiler by adding or subtracting functionality or by applying certain optimizations to the output.

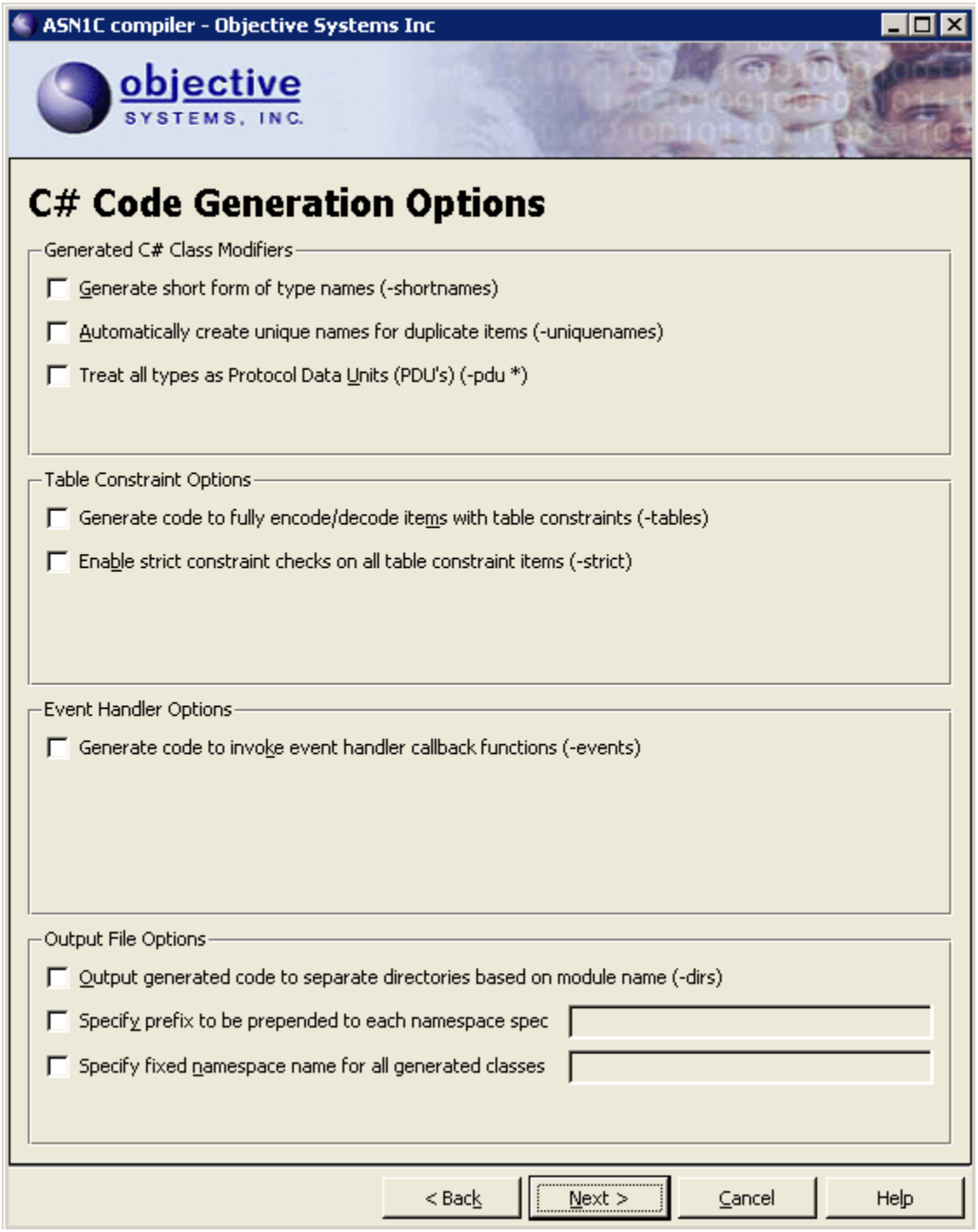
Basic encoding rules are used by default. Only one of BER, DER, and CER can be checked at any time. XML and XER are also mutually exclusive options.

By default, encoding and decoding functions are generated by the compiler. If the target application does not require encoding or decoding capabilities (for example, if it is only intended to read messages and does not need to write them), unchecking the corresponding checkbox will reduce the amount of code generated.

Other functions may also be generated if desired: *Compare*, *Test*, and *Print* all supply extra functionality or control of generated types.

Check *Stream* to modify generated encode and decode functions to use streams instead of memory buffers. This allows encoding and decoding to a source or sink such as a file or socket. Stream-based encoding and decoding cannot be combined with buffer-based.

To reduce the code footprint, several other options may be selected: *Do not generate indefinite length processing code*, *Do not generate code to save/restore unknown extensions*, *Do not generate code to check constraints*, and *Generate compact code* may all be used to reduce the amount of generated code at the expense of some error checking.


ASN1C compiler - Objective Systems Inc
objective
SYSTEMS, INC.
C# Code Generation Options
Generated C# Class Modifiers
Generate short form of type names (-shortnames)
Automatically create unique names for duplicate items (-uniquenames)
Treat all types as Protocol Data Units (PDU's) (-pdu *)
Table Constraint Options
Generate code to fully encode/decode items with table constraints (-tables)
Enable strict constraint checks on all table constraint items (-strict)
Event Handler Options
Generate code to invoke event handler callback functions (-events)
Output File Options
Output generated code to separate directories based on module name (-dirs)
Specify prefix to be prepended to each namespace spec
Specify fixed namespace name for all generated classes
< Back
Next >
Cancel
Help

Click on ‘Help’ to get a summary of what each of these options do or read the section *Running ASN1C from the Command-line*. All of these items are optional. Some items will be grayed-out if they are not applicable to the encoding rules or language previously selected.

Specify prefix to be prepended to each namespace spec and *Specify fixed namespace name for all generated classes* are mutually exclusive and allow the namespace of the generated code to be changed; by default, it is the same as the module.

Compilation

When all options have been specified, the final screen may be used to execute the compilation command:

Included in the window are the compiler command, an option to save the project, and the output from compilation. Selected options are reflected in the command line.

Click *Finish* to terminate the program. The wizard will ask whether or not to save any changes made, whether a new project has been created or not.

Compiler Configuration File

In addition to command line options, a configuration file can be used to specify compiler options. These options can be applied not only globally but also to specific modules and productions.

A simple form of the Extended Markup Language (XML) is used to format items in the file. This language was chosen because it is fairly well known and provides a natural interface for representing hierarchical data such as the structure of ASN.1 modules and productions. The use of an external configuration file was chosen over embedding directives within the ASN.1 source itself due to the fact that ASN.1 source versions tend to change frequently. An external configuration file can be reused with a new version of an ASN.1 module, but internal directives would have to be reapplied to the new version of the ASN.1 code.

At the outer level of the markup is the <asn1config> </asn1config> tag pair. Within this tag pair, the specification of global items and modules can be made. Global items are applied to all items in all modules. An example would be the <storage> qualifier. A storage class such as dynamic can be specified and applied to all productions in all modules. This will cause dynamic storage (pointers) to be used to any embedded structures within all of the generated code to reduce memory consumption demands.

The specification of a module is done using the <module></module> tag pair. This tag pair can only be nested within the top-level <asn1config> section. The module is identified by using the required <name></name> tag pair or by specifying the name as an attribute (for example, <module name=”MyModule”>). Other attributes specified within the <module> section apply only to that module and not to other modules specified within the specification. A complete list of all module attributes is provided in the table at the end of this section.

The specification of an individual production is done using the <production></production> tag pair. This tag pair can only be nested within a <module> section. The production is identified by using the required <name></name> tag pair or by specifying the name as an attribute (for example, <production name=”MyProd”>). Other attributes within the production section apply only to the referenced production and nothing else. A complete list of attributes that can be applied to individual productions is provided in the table at the end of this section.

When an attribute is specified in more than one section, the most specific application is always used. For example, assume a <typePrefix> qualifier is used within a module specification to specify a prefix for all generated types in the module and another one is used to specify a prefix for a single production. The production with the type prefix will be generated with the type prefix assigned to it and all other generated types will contain the type prefix assigned at the module level.

Values in the different sections can be specified in one of the following ways:

1. Using the <name>value</name> form. This assigns the given value to the given name. For example, the following would be used to specify the name of the “H323-MESSAGES” module in a module section:

```
<name>H323-MESSAGES</name>
```

2. Flag variables that turn some attribute on or off would be specified using a single <name/> entry. For example, to specify a given production is a PDU, the following would be specified in a production section:

```
<isPDU/>
```

3. An attribute list can be associated with some items. This is normally used as a shorthand form for specifying lists of names. For example, to specify a list of type names to be included in the generated code for a particular module, the following would be used:

```
<include types="TypeName1,TypeName2,TypeName3"/>
```

The following are some examples of configuration specifications

```
<asn1config><storage>dynamic</storage></asn1config>
```

This specification indicates dynamic storage should be used in all places where its use would result in significant memory usage savings within all modules in the specified source file.

```
<asn1config>
   <module>
      <name>H323-MESSAGES</name>
      <sourceFile>h225.asn</sourceFile>
      <typePrefix>H225</typePrefix>
   </module>
   ...
</asn1config>
```

This specification applies to module 'H323-MESSAGES' in the source file being processed. For IMPORT statements involving this module, it indicates that the source file 'h225.asn' should be searched for specifications. It also indicates that when C or C++ types are generated, they should be prefixed with the 'H225'. This can help prevent name clashes if one or more modules are involved and they contain productions with common names.

The following tables specify the list of attributes that can be applied at all of the different levels: global, module, and individual production:

Global Level

There are no attributes that are specific to C# that can be specified at the global level.

Module Level

These attributes can be applied at the module level by including them within a <module> section:

Name	Values	Description
<name> </name>	module name	This attribute identifies the module to which this section applies. It is required.
<include types="names" values="names"/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be included in the generated code. By default, the compiler generates code for all types and values within a specification. This allows the user to reduce the size of the generated code base by selecting only a subset of the types/values in a specification for compilation. Note that if a type or value is included that has de-

Name	Values	Description
		pendent types or values (for example, the element types in a SEQUENCE, SET, or CHOICE), all of the dependent types will be automatically included as well.
<include importsFrom=”name” />	ASN.1 module name(s) specified as an attribute list.	This form of the include directive tells the compiler to only include types and/or values in the generated code that are imported by the given module(s).
<exclude types=”names” values=”names”/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be excluded in the generated code. By default, the compiler generates code for all types and values within a specification. This is generally not as useful as in *include* directive because most types in a specification are referenced by other types. If an attempt is made to exclude a type or value referenced by another item, the directive will be ignored.
<sourceFile> </sourceFile>	source file name	Indicates the given module is contained within the given ASN.1 source file. This is used on IMPORTs to instruct the compiler where to look for imported definitions. This replaces the **module.txt** file used in previous versions of the compiler to accomplish this function.
<pkgName>	C# namespace name	Name of the C# namespace associated with this module. This will cause a C# using statement to be generated for the module if this name is not the same as that of the namespace being compiled.
<namespace> </namespace>	namespace name	This is used to specify the namespace name for the given module. By default, Asn1c compiler will use the “http:// www.obj-sys.com” as the module namespace. This option should be used with XML encoding rule (XML) only. Asn1c compiler will ignore this option usage with other encoding rules.

Production Level

These attributes can be applied at the production level by including them within a <production> section:

Name	Values	Description
<name> </name>	module name	This attribute identifies the module to which this section applies. It is required.

Name	Values	Description
<isBigInteger/>	n/a	This is a flag variable (an 'empty element' in XML terminology) that specifies that this production will be used to store an integer larger than the C# long type (64 bits). A C# BigInteger class will be used to hold the value. This qualifier can be applied to either an integer or constructed type. If constructed, all integer elements within the constructed type are flagged as big integers.

Compiler Error Reporting

Errors that can occur when generating source code from an ASN.1 source specification take two forms: syntax errors and semantics errors.

Syntax errors are errors in the ASN.1 source specification itself. These occur when the rules specified in the ASN.1 grammar are not followed. ASN1C will flag these types of errors with the error message 'Syntax Error' and abort compilation on the source file. The offending line number will be provided. The user can re-run the compilation with the '-l' flag specified to see the lines listed as they are parsed. This can be quite helpful in tracking down a syntax error.

The most common types of syntax errors are as follows:

- Invalid case on identifiers: module name must begin with an uppercase letter, productions (types) must begin with an uppercase letter, and element names within constructors (SEQUENCE, SET, CHOICE) must begin with lowercase letters.
- Elements within constructors not properly delimited with commas: either a comma is omitted at the end of an element declaration, or an extra comma is added at the end of an element declaration before the closing brace.
- Invalid special characters: only letters, numbers, and the hyphen (-) character are allowed. Programmers tend to like to use the underscore character (_) in identifiers. This is not allowed in ASN.1. Conversely, C or C# does not allow hyphens in identifiers. To get around this problem, ASN1C converts all hyphens in an ASN.1 specification to underscore characters in the generated code.

Semantics errors occur on the compiler back-end as the code is being generated. In this case, parsing was successful, but the compiler does not know how to generate the code. These errors are flagged by embedding error messages directly in the generated code. The error messages always begin with an identifier with the prefix **'%ASN-',.** A search can be done for this string in order to find the locations of the errors. A single error message is output to stderr after compilation on the unit is complete to indicate error conditions exist.

Generated C# Source Code Overview

A separate C# source file with extension '.cs' is generated for each production encountered within an ASN.1 source file. Every ASN.1 type is mapped to a C# class. This is true even at the lowest levels – types such as BOOLEAN, INTEGER, and NULL all have wrapper classes.

The following items may be present in a generated C# file:

- Using statements
- Namespace specification
- Class declaration
- A tag constant object declaration
- Public member variables
- Constructors
- Public Decode() method
- Public Encode() method
- Other methods
- Inner SAX Handler class (XER only)

Additional specialized items may be present as well depending on the base type of the target production. These specialized items are discussed in the sections on ASN.1 to C# mappings for the various ASN.1 types.

A complete generated C# source file for the 'EmployeeNumber' production within the production within the ASN.1 sample file 'employee.asn' can be found on the following page. The ASN.1 production from which this file was generated is as follows:

```
EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER
```

The generated code is as follows:

```
using System;
using Com.Objsys.Asn1.Runtime;

namespace Test {

   public class EmployeeNumber : Asn1Integer {
      public new readonly static Asn1Tag _TAG =
         new Asn1Tag (Asn1Tag.APPL, Asn1Tag.PRIM, 2);

      public EmployeeNumber () : base()
      {
      }

      public EmployeeNumber (long value_) : base(value_)
      {
      }
```

```
        public override void Decode
           (Asn1BerDecodeBuffer buffer, bool explicitTagging, int implicitLength)
        {
           int llen = (explicitTagging) ?
              MatchTag (buffer, _TAG) : implicitLength;

           base.Decode (buffer, false, llen);
        }

        public override int Encode
        (Asn1BerEncodeBuffer buffer, bool explicitTagging)
        {
           int _aal = base.Encode (buffer, false);

           if (explicitTagging) {
              _aal += buffer.EncodeTagAndLength (_TAG, _aal);
           }

           return (_aal);
        }
     }
  }
```

The following sections discuss the various sections of the generated C# source file.

Namespace Specification

The namespace specification is the second item in the file and is declared using the ‘namespace’ keyword. By default, this is set to the name of the ASN.1 module that is being compiled. However, this can be modified by using the –nspfx and –namespace command line options. The –nspfx option adds the specified prefix before the module name. For example, if an ASN.1 module named ‘Employee’ is being compiled and ‘-nspfx test.’ is specified on the command line, the namespace name in the generated source files would be ‘test.Employee’. The –namespace switch takes this a step further. It allows specification of the full namespace name. In the sample specification above, ‘-nspfx sample_ber.’ was specified on the compiler command line.

Standard *using* statements are added for the ASN1C C# run-time classes and C# utility classes. Using statements may also be added for items imported from other ASN.1 modules if they don’t exist within the namespace being generated.

Class Declaration

Next comes the class declaration. It is of the following form:

```
public class <ProdName> : <BaseClass>
```

<ProdName> is the name of the production in the ASN.1 source file. <BaseClass> is a class from which the type is derived. This can either be a standard run-time or compiler-generated class. In our example, the EmployeeNumber is an INTEGER, so we can directly inherit the Asn1Integer run-time base class. If we had a declaration such as the following:

```
EmployeeSSNumber ::= [APPLICATION 22] EmployeeNumber
```

Our EmployeeSSNumber class would be inherited from the compiler-generated EmployeeNumber class as follows:

```
public class EmployeeSSNumber : EmployeeNumber
```

Note: the preceding example is not true if –compact is specified. In that case, all intermediate classes would be removed so *EmployeeSSNumber* would inherit *Asn1Integer* as in the first case.

Tag Constant

The next item in the generated source file is a tag constant. This is only generated if the production is tagged. The runtime class *Asn1Tag* is used for this constant. This class contains methods for operating on ASN.1 tag values. In the sample above, the [APPLICATION 2] tag that is present in the ASN.1 production definition is represented by the generated tag constant.

Public Member Variables

The next section of the file would be public member variables. In our example above, no member variables are present. This is because INTEGER is a primitive type, so the member variable in which the integer value is stored can be found in the *Asn1Integer* base class from which this class is derived. This is true for all primitive types – the value will be contained within the run-time base class.

Constructed types will contain public member variables to represent the elements that make up the type. For example, the following SEQUENCE production:

```
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
   givenNameIA5String,
   initial IA5String,
   familyNameIA5String
}
```

will result in the following public member variables being added to the generated class:

```
public Asn1IA5String givenName;
public Asn1IA5String initial;
public Asn1IA5String familyName;
```

Note that the member variables are public. They were declared this way to make access easier. A trade-off existed between ease-of-use and secure encapsulation. The ease-of-use approach was chosen because it was felt that the repeated use of get/set methods within deeply nested structures would be too clumsy and bulky in most applications. Therefore, the variables were made public to make the encapsulated values easier to set and retrieve. Consistency checks have been added in some methods to make sure values of the correct types are specified for these elements. These checks are discussed in the sections on the specific constructed types.

Constructors

Constructors are generated to allow an object to be initialized in a number of different ways. All productions have a default constructor with no parameters. This creates an empty object that can be filled in at a later time. Constructors are also created that take a parameter of the base type value to allow direct population upon creation of an object. In our example code, two constructors were generated:

```
public EmployeeNumber () : base() { }

public EmployeeNumber (int value_) : base (value_) {
}
```

More complex constructed ASN.1 types such as a SEQUENCE would have a constructor that would have an argument for each defined element. A CHOICE on the other hand would have a unique constructor for each of the possible choice items. See the sections on specific ASN.1 types to find out exactly what constructors are generated for a given type.

Decode Method

The generated decode method for BER/DER has the following general form:

```
public override void Decode (Asn1BerDecodeBuffer buffer,
                             bool explicitTagging, int implicitLength);
```

Users of the C and C++ version of the product might recognize this form. It is very similar to the C function prototype. A reference to an *Asn1BerDecodeBuffer* object is passed that specifies the message being decoded. This is similar to the context variable in the C version of the product.

The *explicitTagging* and *implicitLength* arguments should be of no concern to the average user. The *explicitTagging* argument should be set to `true` and the *implicitLength* argument set to zero. These arguments are only used in internal calls generated by the compiler when implicit tagging is used. In this case, the decoder will at times only be concerned with decoding the contents of a field and not the tag information. At the outer levels, it will always be necessary to decode a tag and length.

The C# decode method reports errors by throwing exceptions. This is a change from the C/C++ version that returned a status value. The *Asn1Exception* class is the base class for all exceptions defined for ASN1C. A complete list of these exceptions can be found in the ASN1C Exceptions section.

For PER, the signature is similar:

```
public override void Decode (Asn1PerDecodeBuffer buffer);
```

In this case, the *explicitTagging* and *implicitLength* arguments are not required since PER has no tagging. The only required argument is a reference to a decode buffer object.

For XER, two overloaded decode methods are generated:

```
public override void Decode (System.Object reader, string xmlURI);

public override void Decode (System.Object reader, Stream byteStream);
```

These take as arguments an XML reader object reference and a reference to an input source object. The XML reader object is a standard class within an XML parser that reads and parses an XML document. The input source can either be a URI (this can be a local filename) or an in-memory byte stream.

Encode Method

The generated encode method for BER/DER has the following general form:

```
public override int Encode (Asn1BerEncodeBuffer buffer, bool explicitTagging);
```

The *Asn1BerEncodeBuffer* argument specifies the buffer into which the message will be encoded. The *explicitTagging* argument is primarily for use by the compiler for generating internal calls to handle implicitly tagged elements in constructed types. Users should always set this argument to `true`.

The encode method returns the length of the encoded component. Unlike the C /C++ version, this return value does not double as a status value as well. Any errors that occur in the encode process are reported by throwing an ASN1C exception. A complete list of these exceptions can be found in the ASN1C Exceptions section.

The general form of a PER encode method is as follows:

```
public override void Encode (Asn1PerEncodeBuffer buffer);
```

In this case, the *explicitTagging* argument is not required since PER has no tagging. The only required argument is a reference to an encode buffer object. Also note that the return value is void instead of int. No intermediate lengths are returned during the encoding of a PER message. Any errors that occur are reported as an exception; hence there is no need for a return value.

The general form of an XER encode method is as follows:

```
public override void Encode (Asn1XerEncodeBuffer buffer, string elemName);
```

In this case, the buffer reference is to an XER encode buffer and an element name argument is added. The element name is the name of the element that is to bracket the XML encoded value (i.e. <elemName>value</elemName>). The return type is also void as in the PER case because errors are reported through the exception mechanism.

Other Methods

Other generated methods include the following:

A private *Init()* method. This is generated in constructed types to set all element object references to null prior to decoding.

A public *Print()* method. This is only generated if the –print option is specified. This provides a formatted printout of the contents of the object. The output can be directed to a StreamWriter object.

A public *ElemName* property (CHOICE only). This property retrieves the name of an element within a CHOICE construct give its assigned identifier value.

The public *Set_<element>* methods (CHOICE only). These are generated for each element in a CHOICE construct to allow the CHOICE value to be set.

Inner Classes

The generation of code for XER causes the following inner class definition to be generated:

```
public class SaxHandler : Asn1XerSaxHandler {
   Asn1XerSaxHandler mElemSaxHandler;
   StringBuffer mCurrElemValue;

   internal SaxHandler() {
      <code ..>
   }

   public void StartElement (string namespaceURI, string localName,
                             string qName, Attributes atts)
   {
      <code ..>
   }

   public void Characters (char[] ch, int start, int length)
   {
      <code ..>
```

```
      }

      public void EndElement (string namespaceURI,
                              string localName, string qName)
      {
         <code ..>
      }
   }
```

This is an implementation of a standard SAX content handler class. As the XML parser software parses messages, the methods within this class are invoked with the parsed content. The *StartElement* method is invoked after a start element tag (<tag>) is parsed. The *Characters* method is invoked one or more times to pass the content between tags into the application. The *EndElement* method is invoked when an end element tag (</tag>) is encountered.

The ASN1C compiler generates custom code for each ASN.1 type within a given specification to parse the XML contents and fill in the generated C# objects.

ASN.1 Type to C# Class Mappings

The following sections discuss the specific mappings of ASN.1 and XSD types to C# classes.

BOOLEAN

The ASN.1 BOOLEAN type is converted to a C# class that inherits the *Asn1Boolean* run-time class. This base class encapsulates the following public member variable:

```
public bool mValue;
```

This is where the Boolean value to be encoded is stored. It also contains the result of a decode operation. Since it is public, it can be accessed directly to get or set the value. The generated constructors can also be used to set the value.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	<name> ::= BOOLEAN
XSD Type:	<xsd:boolean>

Generated C# class:	public class <name> : Asn1Boolean {
	public <name> () : base() {
	}
	public <name> (bool value_) : base (value_) {
	}
	}

This definition assumes a simple assignment of the form “<name> ::= BOOLEAN” (i.e., no tagging or subtypes have been added to the BOOLEAN declaration). In this case, no specific encode or decode methods are generated – calls to these methods pass through to the generic calls defined in the base class. This is true of all other primitive type declarations as well unless otherwise noted.

INTEGER

The ASN.1 INTEGER type is converted to a C# class that inherits the *Asn1Integer* run-time class. This base class encapsulates the following public member variable:

```
public long mValue;
```

This is where the integer value to be encoded is stored. It also contains the result of a decode operation. Since it is public, it can be accessed directly to get or set the value. The generated constructors can also be used to set the value.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	<name> ::= INTEGER
XSD Types:	<xsd:integer>, <xsd:byte>, <xsd:short>, <xsd:int <xsd:long>, <xsd:unsignedByte>, <xsd:unsignedSho <xsd:unsignedInt>, <xsd:unsignedLong>, <xsd:posi <xsd:nonPositiveInt>, <xsd:negativeInt>, <xsd:nonNegativeInt>
Generated C# class:	public class <name> : Asn1Integer { public <name> () : base() {

	}
	public <name> (**long** value_) : **base**(value_) {
	}
	}

This shows the class generated for a simple INTEGER assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

Large Integer Support

The maximum size for a C# long integer type is 64 bits. ASN.1 has no such limitation on integer sizes and some applications (security key values for example) demand larger sizes. In order to accommodate these types of applications, the ASN1C compiler allows an integer to be declared a "big integer" via a configuration file variable (the <isBigInteger/ > setting is used to do this – see the section describing the configuration file for full details). When the compiler detects this setting, it will declare the integer class to be derived from the *Asn1BigInteger* class instead of the *Asn1Integer* class. The *Asn1BigInteger* class encapsulates an object of the C# *BigInteger* class. This provides full support for working with integers of arbitrary lengths.

For example, the following INTEGER type might be declared in the ASN.1 source file:

```
SecurityKeyType ::= [APPLICATION 2] INTEGER
```

Then, in a configuration file used with the ASN.1 definition above, the following declaration can be made:

```
<production>
   <name>SecurityKeyType</name>
   <isBigInteger/>
</production>
```

This will cause the compiler to generate the following class header:

```
class SecurityKeyType : Asn1BigInteger
```

The value field is populated by creating a C# BigInteger object and either passing it in through the constructor or using it to directly populate the public member variable named **mValue** declared in the base class.

BIT STRING

The ASN.1 BIT STRING type is converted to a C# class that inherits the *Asn1BitString* run-time class. This base class encapsulates the following two public member variables:

```
public int numbits;
public byte[] mValue;
```

These describe the bit string to be encoded or decoded.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	<name> ::= BIT STRING

Generated C# class:	**public class** <name> : Asn1BitString {
	public <name> () :
	base() {
	}
	public <name> (**int** numbits_, **byte**[] data) :

```
      base (numbits_, data) {
   }
   public <name> (bool[] bitValues) :
      base (bitValues) {
   }
   public <name> (string value_) :
      base (value_) {
   }
}
```

This shows the class generated for a simple BIT STRING assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

The constructors generated for this type provide additional options for populating the member variables in the base class. In addition to passing the string using the numbits and data arguments to specify a bit string in native format, the string can be specified as an array of boolean values or as a string. The string form expects the string to be passed in the ASN.1 value notation format for either a binary string (i.e., 'xxxx'B) or a hexadecimal string (i.e., 'xxxx'H).

Named Bits

In the ASN.1 standard, it is possible to define an enumerated bit string that specifies named constants for different bit positions. ASN1C provides support for this type of construct by generating symbolic constants that can be used to set, clear, or test these named bits. These symbolic constants are simply the bit names and values in the following general form:

```
public readonly static int <name> = <value>;
```

The base class contains the following methods for using these generated constants:

Set : This method can be used to set a bit in the bit string to be set. There is also an overloaded version that takes a boolean value argument that can be used to set the bit to the given boolean value.

Clear : This method can be used to clear the named bit in the bit string.

Get : This method can be used to test if the named bit is set or clear.

See the *Asn1BitString* class description in the run-time section for more details on these methods.

OCTET STRING

The ASN.1 OCTET STRING type is converted to a C# class that inherits the *Asn1OctetString* run-time class. This base class encapsulates the following public member variable:

```
public byte[] mValue;
```

The number of octets to be encoded or that were decoded is specified in the built-in length component of the array object (i.e., mValue.length).

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	`<name> ::= OCTET STRING`
XSD Types:	`<xsd:hexBinary>, <xsd:base64Binary>`
Generated C# class:	`public class <name> : Asn1OctetString {`

	`public <name> () : base() {`
	`}`
	
	`public <name> (byte[] data) : base (data) {`
	`}`
	
	`public <name> (byte[] data,`
	`int offset,`
	`int nbytes) :`
	`base (data, offset, nbytes) {`
	`}`
	
	`public <name> (string value_) : base (value_) {`
	`}`
	`}`

This shows the class generated for a simple OCTET STRING assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

The constructors generated for this type provide additional options for populating the member variables in the base class. In addition to passing the string directly using the data argument, the string form can be used. The string is passed in ASN.1 value notation format for either a binary string (i.e., 'xxxx'B), hexadecimal string (i.e., 'xxxx'H), or a character string (i.e., 'xxxx'). A constructor also exists that allows a portion of a byte array starting at a given offset and consisting of a given number of bytes to be used to populate the variable.

Character String Types

The C# version of the compiler contains support for the various ASN.1 character string types including the BMP, Universal and UTF-8 string types. All character strings in C# are based on 16-bit Unicode characters except for UniversalString which is based on a 32-bit character set.

All character string types are derived from the *Asn1CharString* base class (except the UniversalString). This class contains the following public member variable that holds the character string contents:

```
public string mValue;
```

Each of the specific ASN.1 character string types except UniversalString has an associated C# class that is derived from the *Asn1CharString* base class. The general form of the C# class name for each of the ASN.1 string types is *Asn1* followed by the ASN.1 string type name. For example, *IA5String* is represented by the *Asn1IA5String* class, *NumericString* by the *Asn1NumericString*, etc.

The UniversalString associated C# class is derived from Asn1Type and it contains the following public member that holds the character string contents:

```
public int mValue[];
```

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	`<name> ::= <CharStrType>`
XSD Types:	<xsd:string> and all related types including date/time types and duration.
Generated C# class:	`public class <name> : Asn1<CharStrType> {` `public <name> () :`

	`base() {`
	`}`
	`public <name> (string value_) :`
	`base (value_) {`
	`}`
	`}`

ENUMERATED

The ASN.1 ENUMERATED type is converted into a C# class that inherits the *Asn1Enumerated* run-time class. In version 6.1, the generated code was changed to conform to Joshua Bloch's static enumeration pattern (as explained in *Effective Java*). Enumerated values are created as singletons to allow for lazy initialization. A specially named object, dec, is created to hold decoded values. In combination, these changes improve application performance, since only a fixed number of objects are allocated for any execution of the application.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	`<name> ::= ENUMERATED { <e1>, <e2>, …, <en> }`
XSD Types:	Any type with an <xsd:enumeration> restriction.

Generated C# class:	`public class <name> : Asn1Enumerated {`
	`private static <name> _<e1> = null;`
	`private static <name> _<e2> = null;`
	…
	`private static <name> _dec = new <name>(-1);`
	
	`protected <name> (int value_) :`
	`base (value_) { }`
	
	`public static <name> <e1>() {`
	`if (_<e1> == null) _<e1> = new <name>(<v1>)`
	
	`return _<e1>;`
	`}`
	…
	
	`public static <name> dec() { return _dec; }`
	
	`public static <name> ValueOf { <code> }`
	
	`public override void Decode () { <code> }`
	`public override int Encode () { <code> }`
	`public override void Print () { <code> }`
	`}`

Notes:

1. The "…" notation used in the ASN.1 definition above does not represent the ASN.1 extensibility notation. It is used to show a continuation of the enumerated sequence of values.

2. The <e1>, <e2>, etc. items denote enumerated constants. These can be in identifier only format or identifier(value) format. The <v1>, <v2>, etc. items denote the enumerated values. These are sequential numbers starting at zero if no values are provided. Otherwise, the actual enumerated values are used.

3. The public methods that are generated are shown without arguments or function bodies for brevity.

In the case of the enumerated type, encode/decode methods are always generated. These verify that the given value is within the defined set. An *Asn1InvalidEnumException* is thrown if the value is not in the defined set unless the enumeration is extensible. In this case, no exception is thrown.

If an extensibility marker (…) is present in the ASN.1 definition, it will not affect the generated constants. A constant will be generated for all options – both root and extended. However, in the *ValueOf* method, an "undefined" constant will be returned to indicate that the value is not in the original specification.

NULL

The ASN.1 NULL type is converted into to a C# class that inherits the *Asn1Null* run-time class. This base class does not contain a public member variable for a value because the NULL type has no associated value.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	`<name> ::= NULL`

Generated C# class:	`public class <name> : Asn1Null {`
	`  public <name> () :`
	`  base() {`
	`  }`
	`}`

This shows the class generated for a simple NULL assignment. If a tagged type is specified, specific encode and decode methods will be generated as well.

OBJECT IDENTIFIER

The ASN.1 OBJECT IDENTIFIER type is converted to a C# class that inherits the *Asn1ObjectIdentifier* run-time class. This base class encapsulates the following public member variable:

```
public int[] mValue;
```

The number of subidentifiers to be encoded or that were decoded is specified in the built-in length component of the array object (i.e., mValue.length).

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	`<name> ::= OBJECT IDENTIFIER`

Generated C# class:	`public class <name> : Asn1ObjectIdentifier {`
	`public <name> () :`
	`base() {`
	`}`
	`public <name> (int[] value_) :`
	`base (value_) {`
	`}`
	`}`

This shows the class generated for a simple OBJECT IDENTIFIER assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

RELATIVE-OID

The ASN.1 RELATIVE-OID type is converted to a C# class that inherits the *Asn1RelativeOID* run-time class. This class inherits the *Asn1ObjectIdentifier* class defined above. The storage of the relative OID value is the same as described for OBJECT IDENTIFIER. The only difference is the extended class defines different implementations of the encode/ decode methods that apply the rules associated with the RELATIVE-OID type.

REAL

The ASN.1 REAL type is converted to a C# class that inherits the *Asn1Real* run-time class. This base class encapsulates the following public member variable:

```
public double mValue;
```

This is where the real value to be encoded is stored. It also contains the result of a decode operation. Since it is public, it can be accessed directly to get or set the value. The generated constructors can also be used to set the value.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	<name> ::= REAL
XSD Types:	<xsd:float>, <xsd:double>

Generated C# class:	public class <name> : Asn1Real {
	public <name> () :
	base() {
	}
	public <name> (double value_) :
	base (value_) {
	}
	}

This shows the class generated for a simple REAL assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

REAL (Base 10)

The ASN.1 Base 10 REAL type is converted to a Java class that extends the *Asn1Real10* run-time class. A base 10 real is specified in ASN.1 using a WITH COMPONENTS clause such as the following:

```
REAL(WITH COMPONENTS {
   ...,
   base (10)
})
```

It is also used for XSD decimal type specifications.

In this case, the real number is stored as a C# character string in the character string base class:

```
public String mValue;
```

ASN.1 Production:	<name> ::= REAL (WITH COMPONENTS { base(10) })

XSD Types:	`<xsd:decimal>`

Generated C# class:	`public class <name> : Asn1Real10 {`
	`public <name> () {`
	`base();`
	`}`
	`public <name> (String value_) {`
	`base (value_);`
	`}`
	`}`

SEQUENCE

The ASN.1 SEQUENCE type is converted to a C# class that inherits the *Asn1Type* run-time base class. Public member variables are generated for each of the elements defined in the SEQUENCE. Each of these member variables represents an object reference since all of the ASN.1 types are mapped to C# objects.

ASN.1 Production:	`<name> ::= SEQUENCE {`
	`<element1-name> <element1-type>,`
	`<element2-name> <element2-type>,`
	`...`
	`}`

XSD Types:	`<xsd:sequence>, <xsd:all>`

Generated C# class:	`public class <name> : Asn1Type {`
	`public <type1> <element1-name>`
	`public <type2> <element2-name>`
	`...`
	`public <name> () : base()`
	`{`
	
	`}`
	`public <name> (<type1> <arg1>, <type2> <arg2>, …) :`
	`base()`
	`{`
	`<element1-name> = <arg1>;`
	`<element2-name> = <arg2>;`
	`...`
	`}`
	`public <name> (<basetype1> <arg1>,`
	`<basetype2> <arg2>, …) :`
	
	`base()`
	`{`
	`<element1-name> = new <type1> (<arg1>);`
	`<element2-name> = new <type2> (<arg2>);`
	`...`
	`}`
	
	`private void Init () { … }`
	`public override void Decode () { … }`
	`public override int Encode () { … }`
	`public override void Print () { … }`

Notes:

1. The … notation used in the ASN.1 definition above does not represent the ASN.1 extensibility notation. It is used to show a continuation of the sequence elements.

2. The <type1>, <type2>, etc. items denote the equivalent C# types generated from the ASN.1 <element-type1>, <element-type2>, etc. definitions.

3. The public and private methods that are generated are shown without arguments or function bodies for brevity.

The compiler first generates a public member variable for each of the elements defined in the SEQUENCE. The decision was made to make these variables public to make them easier to populate for encoding. The alternative was to use protected or private variables with get/set methods for setting or examining the values. It was felt that this approach would be too cumbersome for setting values in deeply nested constructed types.

A default constructor is then generated followed by overloaded constructors for setting the element values. The first form is simply a direct mapping of each of the element types to a constructor argument. The second form only contains arguments for the required types in the SEQUENCE (i.e. OPTIONAL and DEFAULT elements are omitted). The third form uses the base type of each of the elements as the type for each argument. This makes it possible to construct a SEQUENCE or SET using literal variables instead of always having to create an object. Finally, another variant of this constructor with primitive types is generated for required elements only. It is possible that you will not see all of these variations in a given generated class. It depends on a) whether or not the SEQUENCE or SET contains optional items and b) whether or not it contains primitive data items.

For example, the following shows how a variable of a generated class containing two IA5String elements could be constructed:

```
v1 = new HelloWorld (“hello”, “world”);
```

Without this second form of constructor, the following would need to be done:

```
v1 = new HelloWorld (new Asn1IA5String(“hello”),
                     new Asn1IA5String(“world”));
```

Also note that since all member variables are public, it is not necessary to use any of the argument-based constructors at all. A variable can be created using the default constructor and each of the elements populated directly.

Creation of Temporary Types

Temporary types are created when a SEQUENCE (or any other constructed type) definition contains other embedded constructed types. An example of this is as follows:

```
A ::= SEQUENCE {
          x SEQUENCE {
             a1 INTEGER,
             a2 BOOLEAN
          },
          y OCTET STRING SIZE (10)
       }
```

In this example, the production has two elements – x and y. The nested SEQUENCE x has two additional elements – a1 and a2.

The ASN1C compiler first recursively pulls all of the embedded constructed elements out of the SEQUENCE and forms new temporary types. The names of the temporary types are of the form <name>_<element-name1>_<element-

name2>_ … <element-nameN>. Using this algorithm, the ASN.1 type defined above would be reduced to the following equivalent ASN.1 types:

```
A-x ::= SEQUENCE {
   a1 INTEGER,
   a2 BOOLEAN
}

A ::= SEQUENCE {
   x A-x,
   y OCTET STRING SIZE (10)
}
```

The mapping of the ASN.1 types to C# classes would then be done.

In the case of nesting levels greater than two, all of the intermediate element names are used to form the final name. For example, consider the following type definition that contains three nesting levels:

```
X ::= SEQUENCE {
   a SEQUENCE {
      aa SEQUENCE { x INTEGER, y BOOLEAN },
      bb INTEGER
   }
}
```

In this case, the generation of temporary types results in the following equivalent type definitions:

```
X-a-aa ::= SEQUENCE { x INTEGER, y BOOLEAN }

X-a ::= SEQUENCE { aa X-a-aa, bb INTEGER }

X ::= SEQUENCE { X-a a }
```

Note that the name for the *aa* element type is *X-a-aa*. It contains both the name for *a* (at level 1) and *aa* (at level 2). This is a change from v5.1x and lower where only the production name and last element name would be used (i.e., *X-aa*). The change was made to ensure uniqueness of the generated names when multiple nesting levels are used.

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the OPTIONAL keyword. This indicates that the element is not required in the encoded message.

Optional elements are accounted for in the C# version of the compiler by simply using *null* object references to denote the absence of an element. Remember that even the simplest primitive ASN.1 type definitions are wrapped in a C# class definition. Therefore an object must be created for any type defined as an element within a SEQUENCE.

To populate a SEQUENCE object for encoding that contains optional elements, the special constructor(s) for required elements only can be used. The default constructor also can be used followed by the manual creation and setting of the individual element values. The default constructor will initialize all element object references to null, so only the items to be encoded need be populated.

DEFAULT keyword

The DEFAULT keyword allows a default value to be specified for elements within the SEQUENCE. ASN1C will parse this specification and treat it as it does an optional element. Note that the value specification is only parsed in

simple cases for primitive values. It is up to the programmer to provide the value in complex cases. For BER encoding, a value must be specified be it the default or other value.

For DER or PER, it is a requirement that no value be present in the encoding for the default value. For integer and boolean default values, the compiler automatically generates code to handle this requirement based on the value in the structure. For other values, the default value is handled the same as an optional element (i.e., a *null* object reference indicates that nothing should be transmitted). The programmer must set the element object reference to *null* on the encode side to specify default value selected. If this is done, a value is not encoded into the message. On the decode side, the developer must test for a *null* object reference. If this is the case, the default value specified in the ASN.1 specification is used.

Extension Elements

If the SEQUENCE type contains an open extension field (i.e., a … at the end of the specification or a …, … in the middle), a special element will be inserted to capture encoded extension elements for inclusion in the final encoded message. This element will be of type *ASN1OpenExt* and have the name *extElem1*. This field will contain the complete encoding of any extension elements that may have been present in a message when it is decoded. On subsequent encode of the type, the extension fields will be copied into the new message.

If the SEQUENCE type contains an extension marker and extension elements, then the open extension type field will not be added. Instead, the actual extension elements will be present. These elements will be treated as optional elements whether they were declared that way or not. The reason is because a version 1 message could be received that does not contain the elements.

XSD <xsd:all> Type Mapping

As per the X.694 standard, the XSD all type is mapped to an ASN.1 SEQUENCE type with a special element add named *order*. This is added as a special element to the generated C# class with the name *_order*. This contains an index entry for each element that identifies the order elements are to be serilaized in when encoded in XML. By default, the array is initialized to encode the elements in the same order as specified in the type. When an XML document of this type is decoded, the order in which the elements are received in recorded in this array. If the data is serialized out in binary form (BER or PER) the array is included in the encoding. If is only transparent in XML encode/decode operations to mimic the behavior of its handling in XSD.

An example of how this is used might be a gateway application that read XML data and then translated to binary form for transmission over a low bandwidth network. When received on the other end, the receiving application would transcode back from binary to XML. Suppose the item being transmitted was described using an xsd:all type that had three elements: a, b, and c. When the original XML document was received by the sending application, suppose the elements were received in the order c, b, a. The order array would record this fact and it would be included in the binary serialization. When the receiver decoded the message on the other end, the order information would be available along with the element data. The receiver could then reconstruct the XML document with the items in the same order as received.

SET

The ASN.1 SET type is converted into a C# class that is identical to that for SEQUENCE as described in the previous section. The only difference between SEQUENCE and SET is that elements may be transmitted in any order in a SET whereas they must be in the defined order in a SEQUENCE. The only impact this has on ASN1C is in the generated decoder for a SET type.

The decoder must take into account the possibility of out-of-order elements. This is handled by using a loop to parse each element in the message. Each time an item is parsed, an internal mask bit within the decoder is set to indicate the element was received. The complete set of received elements is then checked after the loop is completed to verify all required elements were received.

SEQUENCE OF

The ASN.1 SEQUENCE OF type is converted to a C# class that inherits the *Asn1Type* run-time base class. An array public member variable named elements is generated to hold the elements of the defined type.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	`<name> ::= SEQUENCE OF <type>`
XSD Types:	Eleements or content group definitions containing the minOccurs and/or maxOccurs facets. Also, <xsd:list> types use this model.

Generated C# class:	`public class <name> : Asn1Type {`
	`  public <type>[] elements`
	
	`  public <type> () {`
	`  elements = null;`
	`  }`
	`  public <type> (int numRecords) {`
	`  elements = new <type> [numRecords];`
	`  }`
	`  public override void Decode () { … }`
	`  public override int Encode () { … }`
	`  public override void Print () { … }`
	`}`

The compiler first generates a public member variable to hold the SEQUENCE OF elements. The decision was made to make the variable public to make it easier to populate for encoding. The alternative was to use protected or private variables with get/set methods for setting or examining the values. It was felt that this approach would be too cumbersome for setting values in deeply nested constructed types.

Two constructors are generated: a default constructor and a constructor that takes a number of elements argument. The default constructor will set the *elements* variable to null. The second constructor will allocate space for the given number of elements. The recommended way to populate a variable of this type for encoding is to use the second form of the constructor to allocate the required number of elements and then directly set the element object values. For example, to populate the following construct:

```
IntSeq ::= SEQUENCE OF INTEGER
```

with 3 integers, the following code could be used:

```
IntSeq intSeq = new IntSeq (3);
intSeq.elements[0] = new Asn1Integer (1);
intSeq.elements[1] = new Asn1Integer (2);
intSeq.elements[2] = new Asn1Integer (3);
```

Note that each of the integer element values is wrapped in an Asn1Integer wrapper class.

Generation of Temporary Types for SEQUENCE OF Elements

As with other constructed types, the <type> variable can reference any ASN.1 type, including other ASN.1 constructed types. Therefore, it is possible to have a SEQUENCE OF SEQUENCE, SEQUENCE OF CHOICE, etc.

When a constructed type is referenced, a temporary type is generated for use in the final production. The format of this temporary type name is as follows:

```
<prodName>_element
```

In this definition, <prodName> refers to the name of the production containing the SEQUENCE OF type.

For example, a simple (and very common) single level nested SEQUENCE OF construct might be as follows:

```
A ::= SEQUENCE OF SEQUENCE { INTEGER a, BOOLEAN b }
```

In this case, a temporary type is generated for the element of the SEQUENCE OF construct. This results in the following two equivalent ASN.1 types:

```
A-element ::= SEQUENCE { INTEGER a, BOOLEAN b }

A ::= SEQUENCE OF A-element
```

These types are then converted into the equivalent C# classes using the standard mapping that was previously described.

SEQUENCE OF Type Elements in Other Constructed Types

Frequently, a SEQUENCE OF construct is used to define an array of some common type in an element in some other constructed type (for example, a SEQUENCE). An example of this is as follows:

```
SomePDU ::= SEQUENCE {
   addresses SEQUENCE OF AliasAddress,
   ...
}
```

Normally, this would result in the *addresses* element being pulled out and used to create a temporary type with a name equal to “SomePDU-addresses” as follows:

```
SomePDU-addresses ::= SEQUENCE OF AliasAddress

SomePDU ::= SEQUENCE {
   addresses SomePDU-addresses,
   ...
}
```

However, when the SEQUENCE OF element references a simple defined type as above with no additional tagging or constraint information, an optimization is done to cut down on the size of the generated code. This optimization is to generate a common name for the new temporary type that can be used for other similar references. The form of this common name is as follows:

```
_SeqOf<elementProdName>
```

So instead of this:

```
SomePDU-addresses ::= SEQUENCE OF AliasAddress
```

The following equivalent type would be generated:

```
_SeqOfAliasAddress ::= SEQUENCE OF AliasAddress
```

The advantage is that the new type can now be easily reused if “SEQUENCE OF AliasAddress” is used in any other element declarations. Note the (illegal) use of an underscore in the first position. This is to ensure that no name collisions occur with other ASN.1 productions defined within the specification.

An example of the savings of this optimization can be found in H.225. The above element reference is repeated 25 different times in different places. The result is the generation of one new temporary type that is referenced in 25 different places. Without this optimization, 25 unique types with the same definition would have been generated.

SET OF

The ASN.1 SET OF type is converted into a C# class that is identical to that for SEQUENCE OF as described in the previous section.

CHOICE

The ASN.1 CHOICE type is converted to a C# class that inherits the *Asn1Choice* run-time base class. This base class contains protected member variables to hold the choice element object and a selector value to specify which item in the CHOICE was chosen. Methods are generated to get and set the base class members.

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	<name> ::= CHOICE {
	  <element1-name> <element1-type>,
	  <element2-name> <element2-type>,
	  ...
	}

XSD Types:	<xsd:choice>, <xsd:union>

Generated C# class:	public class <name> : Asn1Choice {
	public const byte _<ELEMENT1-NAME> 1
	public const byte _<ELEMENT2-NAME> 2
	...
	
	public <name> () : base() { }
	
	public override string ElemName { … }
	
	public void Set_<element1-name> () { … }
	public void Set_<element2-name> () { … }
	...
	public override void Decode () { … }
	public override int Encode () { … }
	public override void Print () { … }
	}

Notes:

1. The … notation used in the ASN.1 definition above does not represent the ASN.1 extensibility notation. It is used to show a continuation of the sequence elements.

2. The public and private methods that are generated are shown without arguments or function bodies for brevity.

The compiler generates sequential identification constants for each of the defined elements in the CHOICE construct. The format used is the element names converted to all uppercase characters and preceded by an underscore. The

constants represent the values returned by the base class *ChoiceID* property can therefore be used to determine what type of choice element was received in a decode operation.

The *ElemName* property is generated by the compiler and returns the name of the selected element.

A series of *Set_<element>* methods are generated for setting the element value. In these declarations, <element> would be replaced with the actual element names. This is the only way an element value can be set for encoding; these methods ensure a consistent setting of both the element identifier and object reference values.

To access the value of a generated CHOICE object, the *ChoiceID* property and *GetElement* methods within the base class are used. This is generally done with an if or switch statement as follows:

```
Asn1BMPString element;
   if (aliasAddress.ChoiceID == AliasAddress._H323_ID) {
element = (Asn1BMPString) aliasAddress.GetElement();
}
```

In this case, *ChoiceID* is invoked and the result tested to see if the expected value was received. If it was, the element is assigned using *GetElement* with a cast operation.

Creation of Temporary Types

The rules for the generation of CHOICE temporary type variables are the same as they were for SEQUENCE and SET variables. Complex nested types are pulled out of the definitions and used to create additional types to reduce the nesting levels. An example of this is as follows:

```
TestChoice ::= CHOICE {
   a    INTEGER,
   b    BOOLEAN,
   c    SEQUENCE { aa IA5String, bb NULL }
}
```

This would be reduced to the following equivalent ASN.1 productions:

```
TestChoice-c ::= SEQUENCE { aa IA5String, bb NULL }

TestChoice ::= CHOICE {
   a    INTEGER,
   b    BOOLEAN,
   c    TestChoice-c
}
```

this case, the embedded constructed element for option *c* was pulled out to form the *TestChoice-c* production and then this new production is referenced in the original definition.

Populating Generated Choice Structures for Encoding

The only way a CHOICE construct can be populated for encoding is by using one the generated *Set_<element>* methods. It is necessary to do it this way because the base class contains two protected member variables (*choiceID* and element) that must be set consistently. This is the only instance of a mapped type where the mapped element values do not have public access.

The following demonstrates setting a variable of the TestChoice structure defined above to use the first option:

```
TestChoice testChoice = new TestChoice ();
```

```
testChoice.Set_a (new Asn1Integer (222));
```

Accessing the Choice Element Value after Decoding

To access the element in a choice construct after decoding, the following two methods can be used (both are defined in the *Asn1Choice* base class):

1. *ChoiceID* – this returns an identifier equal to one the generated choice identifier constants, and
2. *GetElement* – this returns a reference to the decoded element object. It is of type *Asn1Type* but it can be upcast to the correct element type using information from the *ChoiceID* call.

In addition, the compiler generates a *ElemName* property that can be used to get the textual name of the decoded element.

XSD <xsd:union> Type Mapping

The <xsd:union> type is handled in a similar fashion to a choice type. The main difference is that the items in a union are not tagged. As per X.694, special element names are generated for these items for use in an ASN.1 CHOICE type. These names are based on the base name *alt* and progress with sequential digits added for each addional union item (*alt-1*, *alt- 2*, etc.). XML decoding is accomplished by attempting to decode the content of each alternative in the union and setting the value to the first alternative that can be decoded successfully.

Open Type

Note: The X.680 Open Type replaces the X.208 ANY or ANY DEFINED BY constructs. An ANY or ANY DEFINED BY encountered within an ASN.1 module will result in the generation of code corresponding to the Open Type described below.

The ASN.1 Open Type is converted into a C# class that inherits the *Asn1OpenType* class. This class in turn inherits the *Asn1OctetString* class and provides the following public member variable for storing the encoded message component:

```
public byte[] mValue;
```

The number of octets to be encoded or that were decoded is specified in the built-in length component of the array object (i.e., mValue.length).

The following shows the basic mapping from ASN.1 type to C# class definition:

ASN.1 Production:	<name> ::= <openType>

Generated C# class:	public class <name> : Asn1OpenType {
	public <name> () : base() { }
	
	public <name> (byte[] data) : base (data) {
	}
	
	public <name> (byte[] data,
	int offset,
	int nbytes) :
	base (data, offset, nbytes)
	{
	}

```
  public <name> (Asn1EncodeBuffer buffer) :
     base ()
  {
  }
}
```

The <openType> placeholder is to be replaced with any type of open type specification. It could be the ANY or ANY DEFINED BY keywords from the X.208 specification or an open type from X.681 (for example, TYPEIDENTIFIER.& Type).

The last form of the constructor shown above is for an optimized form of Open Type encoding. When encoding is done using BER, an open type header can be directly added to the beginning of an encoded message component. By using this form of the constructor, you are indicating to the run-time encoder that the encoded message component onto which a header is to be added is already present in the message buffer. The advantage is that binary copies of the encoded message components are avoided both from the encode buffer to the open type object and from the open type object back to the encode buffer.

For XER, a new class derived from the *Asn1OpenType* class was created. This is the Asn1XerOpenType class and this must be used whenever an open type is required for XER. The reason for creating a special derived class is because of dependencies on XML parser classes defined within this class. If these were added directly to the *Asn1OpenType* class, a user would need to always have XML parser files included in their classpath – even if working with BER, DER, or PER only.

If the –tables command line option is selected and the ASN.1 type definition references a table constraint, the code generated is different. In this case, *Asn1OpenType* above is replaced with *Asn1Type*. This the base class for all ASN.1 types. This allows a value of any ASN.1 type to be specified. On the encoding side, a user can assign an object of any ASN.1 type to this variable and the encoding routine will call the appropriate encoder according to the table index value. If the variable type is not present in the table and the Object Set is extensible, than it can be encoded as an open type. Otherwise an exception will be thrown. On the decoding side, the appropriate variable type is populated from the table based on the decoded index parameters. The user can determine the variable type from the table index value. If the variable type is not present in table, then it will be decoded as an open type if the Object Set is extensible; otherwise and exception will be thrown.

<xsd:any> Handling

The XSD any wildcard item is similar to an ASN.1 open type in semantics in that it allows any valid content to be present in that position in an XML document. However, an ASN.1 open type is not used to model an <xsd:any>. Instead, a character string variable is used. This stores the full XML text of the field in native XML form (i.e. angle brackets and the like are not escaped). Note that the XML text is not converted to different form when using binary encoding rules - it is maintained as XML text.

External Type

The ASN.1 EXTERNAL type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. The type is described using the following ASN.1 SEQUENCE:

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE {
   direct-reference OBJECT IDENTIFIER OPTIONAL,
   indirect-reference INTEGER OPTIONAL,
   data-value-descriptor ObjectDescriptor OPTIONAL,
   encoding CHOICE {
      single-ASN1-type  [0] ANY,
      octet-aligned     [1] IMPLICIT OCTET STRING,
      arbitrary         [2] IMPLICIT BIT STRING
```

```
    }
}
```

The ASN.1 compiler is used to create a meta-definition for this structure. The definition will be generated in the file *Asn1External.cs* (or *Asn1XerExternal.cs* for XER), if needed by compiling ASN.1 file. An object created from the resulting C# class is populated just like any other compiler-generated structure for working with ASN.1 data.

EmbeddedPDV Type

The ASN.1 EMBEDDED PDV type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. It was introduced in 1994 to replace EXTERNAL by removing unneeded fields and adding a few new ones to hold information that was missing. This type is described using the following ASN.1 SEQUENCE:

```
EmbeddedPDV ::= [UNIVERSAL 11] IMPLICIT SEQUENCE {
   identification CHOICE {
      syntaxes SEQUENCE {
         abstract OBJECT IDENTIFIER,
         transfer OBJECT IDENTIFIER },
      syntax OBJECT IDENTIFIER,
      presentation-context-id INTEGER,
      context-negotiation SEQUENCE {
         presentation-context-id INTEGER,
         transfer-syntax OBJECT IDENTIFIER },
      transfer-syntax OBJECT IDENTIFIER,
      fixed NULL
   },,
   data-value OCTET STRING }
( WITH COMPONENTS {
            ... ,
            data-value-descriptor ABSENT } )
```

The ASN.1 compiler is used to create a meta-definition for this structure. The definition will be generated in the file *Asn1EmbeddedPDV.cs* (or *Asn1XerEmbeddedPDV.cs* for XER), if needed by compiling ASN.1 files. An object created from the resulting C# class is populated just like any other compiler-generated structure for working with ASN.1 data.

Parameterized Types

The ASN1C compiler can parse parameterized type definitions and references as specified in the X.683 standard. These types allow dummy parameters to be declared that will be replaced with actual parameters when the type is referenced. This is similar to templates in C++.

A simple and common example of the use of parameterized types is for the declaration of an upper bound on a sized type as follows:

```
SizedOctetString{INTEGER:ub} ::= OCTET STRING (SIZE (1..ub))
```

In this definition, ‘ub’ would be replaced with an actual value when the type is referenced. For example, a sized octet string with an upper bound of 32 would be declared as follows:

```
OctetString32 ::= SizedOctetString{32}
```

The compiler would handle this in the same way as if the original type was declared to be an octet string of size 1 to 32. In the case of C#, this would result in size constraint checks being added to the generated encode and decode methods for the type.

Another common example of parameterization is the substitution of a given type inside a common container type. For example, security specifications frequently contain a ‘signed’ parameterized type that allows a digital signature to be applied to other types. An example of this would be as follows:

```
SIGNED { ToBeSigned } ::= SEQUENCE {
   toBeSigned    ToBeSigned,
   algorithmOID  OBJECT IDENTIFIER,
   paramS        Params,
   signature     BIT STRING
}
```

An example of a reference to this definition would be as follows:

```
SignedName ::= SIGNED { Name }
```

where ‘Name’ would be another type defined elsewhere within the module.

ASN1C performs the substitution to create the proper C# class definition for SignedName:

```
public class SignedName : Asn1Type {
  public Name toBeSigned;
  public Asn1ObjectIdentifier algorithmOID;
  public Params paramS;
  public Asn1BitString signature;
  ...
}
```

When processing parameterized type definitions, ASN1C will first look to see if the parameters are actually used in the final generated code. If not, they will simply be discarded and the parameterized type converted to a normal type reference. For example, when used with information objects, parameterized types are frequently used to pass information object set definitions to impose table constraints on the final type. Since table constraints do not affect the code that is generated by the compiler, the parameterized type definition is reduced to a normal type definition and references to it are handled in the same way as defined type references. This can lead to a significant reduction in generated code in cases where a parameterized type is referenced over and over again.

For example, consider the following often-repeated pattern from the UMTS 3GPP specs:

```
ProtocolIE-Field {RANAP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
   id             RANAP-PROTOCOL-IES.&id          ({IEsSetParam}),
   criticality    RANAP-PROTOCOL-IES.&criticality ({IEsSetParam}{@id}),
   value          RANAP-PROTOCOL-IES.&Value       ({IEsSetParam}{@id})
}
```

In this case, *IEsSetParam* refers to an information object set specification that constrains the values that are passed for any given instance of a type referencing a *ProtocolIE-Field*. The compiler does not add any extra code to check for these values, so the parameter can be discarded. After processing the Information Object Class references within the construct (refer to the *Information Objects* section for information on how this is done), the reduced definition for *ProtocolIE-Field* becomes the following:

```
ProtocolIE-Field ::= SEQUENCE {
   id ProtocolIE-ID,
   criticality Criticality,
   value ASN.1 OPEN TYPE
}
```

References to the field are simply replaced with a reference to the generated *ProtocolID-Field* class.

Value Specifications

The ASN1C compiler can parse any type of ASN.1 value specification, however, the basic version will only generate code for the following types of value specifications:

- BOOLEAN
- INTEGER
- ENUMERATED
- Binary String
- Hexadecimal String
- Character String
- OBJECT IDENTIFER

The Pro version of the compiler will generate code for the following remaining types of value specifications:

- Enumerated
- Real
- Sequence
- Set
- Sequence Of
- Set Of
- Choice

If any of the above types of value specifications are detected in an ASN.1 module, the compiler will generate a C# source file with a special class to hold the values. The name of the source file and class is of the following format:

```
_<ModuleName>Values
```

In this definition, <ModuleName> would be replaced with the name of the ASN.1 module in which the values are defined.

The following sections provide details on the C# constants generated for the various types of ASN.1 value specifications.

INTEGER Value Specification

An INTEGER value specification causes a C# integer constant to be generated. stant to be generated.

ASN.1 production:

```
<name> INTEGER ::= <value>
```

Generated C# constant:

```
public static readonly int <name> = <value>;
```

BOOLEAN Value Specification

A BOOLEAN value specification causes a C# boolean constant to be generated.

ASN.1 production:

```
<name> BOOLEAN ::= <value>
```

Generated C# constant:\

```
public static readonly bool <name> = <value>;
```

Binary String Value Specification

This value specification causes two C# constants to be generated: a ‘numbits’ constant specifying the number of bits in the string and a ‘data’ constant that hold the actual bit values.

ASN.1 production:

```
<name> BIT STRING ::= ‘bbbbbbb’B
```

Generated C# constants:

```
public static readonly int <name>_numbits = <numbits>;
public static readonly byte[] <name>_data = { 0xhh, 0xhh, ... };
```

In the ASN.1 production definition, the lowercase ‘b’s above represent binary digits (1’s or 0’s). The generated code contains a *numbits* constant set to the number of bits (binary digits) in the string. The *data* constant specifies the binary data using hexadecimal byte values.

Hexadecimal String Value Specification

This value specification causes a C# constant to be generated containing a byte array of the hexadecimal byte values.

ASN.1 production:

```
<name> OCTET STRING ::= ‘hhhhhh’H
```

Generated C# constants:

```
public static readonly byte[] <name> = { 0xhh, 0xhh, ... };
```

In the ASN.1 production definition, the lowercase ‘h’s above represent hexadecimal digits (0-9, a-f, or A-F). The generated constant specifies the binary data using hexadecimal byte values.

Character String Value Specification

A character string declaration causes a C# String constant to be generated.

ASN.1 production:

```
<name> <StringType> ::= ‘ccccccc’
```

Generated C# constants:

```
public static readonly string <name> = “ccccccc”;
```

In the ASN.1 production definition, <StringType> would be replaced with one of the ASN.1 character string types (for example, IA5String). The lowercase ‘c’s represent string characters. The generated constant is simply the string in C# form.

Object Identifier Value Specification

An object identifier value specification causes a C# integer array to be generated containing the subidentifier values.

ASN.1 production:

```
<name> OBJECT IDENTIFIER ::= <oidvalue>
```

Generated C# constants:

```
public static readonly int[] <name> = { id1, id2, …, idn };
```

For example, consider the following declaration:

```
oid OBJECT IDENTIFIER ::= { ccitt b(5) 10 }
```

This would result in the following C# constant being generated:

```
public static readonly int[] oid = { 0, 5, 10 };
```

ENUMERATED Value Specification

An ENUMERATED value specification causes a C# integer constant to be generated.

ASN.1 production:

```
<name> <enumtype> ::= <enumitem>
```

Generated C# constants:

```
public static readonly int <name> = <enumvalue>;
```

enumvalue will be the sequential integer value corresponding to the *enumitem* in *enumtype*.

REAL Value Specification

A REAL value specification causes a C# double constant to be generated.

ASN.1 production:

```
<name> REAL ::= <value>
```

Generated C# constants:

```
public static readonlyreadonly double <name> = <value>;
```

SEQUENCE Value Specification

A SEQUENCE value specification causes a readonly static instance of the C# class generated for the SEQUENCE to be generated.

ASN.1 production:

```
<name> <SequenceType> ::= <value>
```

Generated C# constants:

```
public static readonly <SequenceType> <name> =
   new <SequenceType> ( new <Elem1Type> (<elem1value>),
                        new <Elem2Type> (<elem2value>),
                        ... );
```

For example, consider the following declaration:

```
SeqType ::= SEQUENCE {
   oid     OBJECT IDENTIFIER,
   id      INTEGER
}

value SeqType ::= { oid { 0 1 1 }, id 12 }
```

This would result in the following C# constant being generated for value:

```
public static readonly SeqType value = new SeqType (
   new Asn1ObjectIdentifier( new int[]{0, 1, 1}),
   new Asn1Integer(12)
);
```

SET Value Specification

The value code generation for the ASN.1 SET type is that same as that for SEQUENCE described above.

SEQUENCE OF Value Specification

A SEQUENCE OF value specification causes a C# array constant to be generated.

ASN.1 production:

```
<name> <SequenceOfType> ::= <value>
```

Generated C# constants:

```
public static readonly <SequenceOfType> <name> =
   new <SequenceOfType>[] {
   new <ElemType> (<elem1value>),
   new <ElemType> (<elem2value>),
   ... };
```

For example, consider the following declaration:

```
SeqOfType ::= SEQUENCE OF INTEGER

value SeqOfType ::= { 1, 2 }
```

This would result in the following C# constant being generated for value:

```
public static readonly SeqOfType value = new SeqOfType[] {
   new Asn1Integer(1),
   new Asn1Integer(2)
```

```
};
```

SET OF Value Specification

The value code generation for the ASN.1 SET OF type is that same as that for SEQUENCE OF described above.

CHOICE Value Specification

A CHOICE value specification causes a readonly static instance of the C# class generated for the CHOICE to be generated.

ASN.1 production:

```
<name> <ChoiceType> ::= elemname : <elemvalue>
```

Generated C# constants:

```
public static readonly <ChoiceType> <name> =
   new <ChoiceType> (<ElemCode>,
                     new <ElemType> (<elemvalue>));
```

For example, consider the following declaration:

```
ChoiceType ::= CHOICE { oid OBJECT IDENTIFIER, id INTEGER }

value ChoiceType ::= id: 1
```

This would result in the following C# constant being generated:

```
public static readonly ChoiceType value =
   new ChoiceType (ChoiceType._ID, new Asn1Integer(1));
```

Generated BER/DER/CER Encode Methods

Two different types of BER (Basic Encoding Rules) encode methods may be generated using the ASN1C compiler:

- Memory-buffer based definite length backward encoders
- Stream-based indefinite length forward encoders

For DER (Distinguished Encoding Rules), only the first option is available because a requirement of DER is that all lengths must be in definite form. For CER (Canonical Encoding Rules), only the second option is available because all constructed element lengths must be in indefinite length form. Each of these methods are described in the following sections.

Memory-buffer Based Definite Length Encoders

For each ASN.1 production defined in the ASN.1 source file, a C# memory-buffer based encode method may be generated. These are the default type of encode methods that are generated when the -ber command-line option is selected. To generate stream-based forward encoders, the -stream option must be specified as well.

An encode method is only generated if it is required to alter the encoding of the base class method. The C# model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types. These default implementations include support for adding the universal tags associated with the types as defined in the X.680 standard.

So for simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated C# class inherits the *Asn1Integer* base class and the default encode method within this class is sufficient to encode a value of the generated type.

However, if the type is altered to contain a tag or constraint, then a custom encode method would be generated:

```
X ::= [APPLICATION 1] INTEGER
```

In this case, special logic is necessary to apply the tag value.

Some types will always cause encode methods to be generated. At the primitive level, this is true for the ENUMERATED type. This type will always contain a custom set of enumerated values. All constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will cause encode methods to be added to the generated classes.

Generated C# Method Format and Calling Parameters

The signature for a C# BER memory buffer encode method is as follows:

```
public override int Encode (Asn1BerEncodeBuffer buffer, bool explicitTagging)
```

The *buffer* argument is a reference to an *Asn1BerEncodeBuffer* object that describes the buffer into which a message is being encoded. This must be created and initialized before calling any encode method. See the description of this class in the *C# Run-Time Classes* section for details on how this class is used.

The *explicitTagging* argument specifies whether or not an explicit tag should be applied to the encoded contents. The average user will almost always want to set this argument to true. The only time it would not be set to true is if a user wanted to just encode a contents field with no tag. This argument is used primarily by the compiler when generating internal calls to properly handle implicit and explicit tagging.

The return value is the length in octets of the encoded message component. Unlike the C/C++ version, a negative value is never returned to indicate an encoding failure. That is handled by the exception mechanism. All ASN1C C# exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Populating Generated Variables for Encoding

Populating generated variables for encoding can be done in most cases either through the object constructors or directly by assigning an object reference to a public member variable.

Constructors are provided for most generated types to allow direct population of the encapsulated member variable(s) on initialization. The exception is the classes generated for SEQUENCE OF or SET OF. These only allow the size of an array to be specified – population of the array elements must be done manually.

All of the base run-time classes except *Asn1Null* contain public member variables. In practically all cases there is a single variable called *mValue* that is of the base type that needs to be populated. For example, the *Asn1Integer* base class contains the following item:

```
public long mValue;
```

Therefore, population of any class variable derived from INTEGER can be done by adding.*mValue* to the end of the lefthand side of the assignment and an integer value on the right. So for the following assignment:

```
X ::= INTEGER
```

A variable of the type can either be populated using the constructor with the following statement:

```
X x = new X (25);
```

or via direct access of the member variable as follows:

```
X x = new X ();
x.mValue = 25;
```

The only primitive type that does not have a single member called mValue to represent its value is BIT STRING. In this case, the *Asn1BitString* class contains a second variable called numbits to specify the number of bits in the string.

Procedure for Calling Memory-Buffer Based BER Encode Methods

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded.

2. Invoke the encode method.

3. Invoke encode message buffer methods to access the encoded message component.

The first step is the creation of an encode message buffer object. Unlike the C/C++ version of the product, there is no choice to be made between a static or dynamic encode buffer. In C#, everything is dynamic. There are two forms of the constructor: a default constructor and one that allows specification of a message buffer size increment. The size

increment will determine how often the buffer will need to be resized to hold large messages. If you know that you will be encoding large messages, then this object should be constructed with a large value for the increment. If you know that you will be encoding small messages in a constrained environment, then this value can be set very low. The default constructor sets the value to a reasonable mid-range value (see SIZE_INCREMENT in *Asn1EncodeBuffer.cs*, as of this writing the value was set to 1024).

The second step is the invocation of the encode method. The calling arguments were described earlier. As per the C# standard, this method must be invoked from within a try/catch block to catch the possible *Asn1Exception* that may be thrown. Alternatively, the method from which the encode method is called can declare that it throws an *Asn1Exception* leaving it to be dealt with at a higher level.

Finally, encode buffer methods can be called to access the encoded message component. The encode method itself returns the length of the component, so this item is already known (however, there is a *MsgLength* property available if you want to access this length from a different location). Unlike C or C++, a pointer to where the message starts in the encode buffer cannot be returned (recall that BER encoding is done from back to front, so a message rarely starts at the beginning of a buffer). However, the C# API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object therefore provides a property called *ByteArrayInputStream* which is the preferred way to access the encoded component.

In addition to *ByteArrayInputStream* there is a *MsgCopy* property that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. The encode buffer class also contains other methods for operating directly on the encoded component (for example, the *write* method can be used to write it to a file or other medium). And of course, one could derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1BerEncodeBuffer* class in the run-time section for a full description of the available methods.

A complete example showing how to invoke an encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

// Step 1: Create a message buffer object. This object uses the
// default size increment for buffer expansion..

Asn1BerEncodeBuffer encodeBuffer = new Asn1BerEncodeBuffer();

// Step 2: Invoke the encode method. Note that it must be done
// from within a try/catch block..

try {
   personnelRecord.Encode (encodeBuffer, true);

   if (trace) {
      System.Console.Out.WriteLine ("Encoding was successful");
      System.Console.Out.WriteLine ("Hex dump of encoded record:");
      encodeBuffer.HexDump ();
      System.Console.Out.WriteLine ("Binary dump:");
      encodeBuffer.BinDump ();
   }

   // Step 3: Access the encoded message component. In this
   // case, we use methods in the class to write the component
   // to a file and output a formatted dump to the message.dmp
   // file..

   // Write the encoded record to a file
```

```
      encodeBuffer.Write(new System.IO.FileStream(
         filename, System.IO.FileMode.Create));

      // Generate a dump file for comparisons

      System.IO.StreamWriter messagedmp =
         new System.IO.StreamWriter(new System.IO.FileStream(
            "message.dmp", System.IO.FileMode.Create));
      messagedmp.AutoFlush = true;
      encodeBuffer.HexDump(messagedmp);
   }
   catch (Exception e) {
      System.Console.Out.WriteLine (e.Message);
      Asn1Util.WriteStackTrace(e, Console.Error);
      return;
   }
```

Reuse of C# Encoding Objects

The simple example above showed the procedure to encode a single record. But what if you had to decode a series of the same record over and over again? This is a common occurrence in a BER encoding application.

You would not want to recreate the data holder and message buffer objects on each pass of the loop. This would have an adverse effect on the performance of the application. What you would want to do is only create the objects a single time and then reuse them to encode each message instance.

It turns out that this is an easy thing to do. The public member variable access to the data holder object makes it easy to change the variables on each given pass. And the encode buffer object contains a *Reset* method for resetting the encode buffer for subsequent encodings. The use of this method has the advantage of not releasing any of the memory that had been accumulated to this point for previous encodings.

To show an example of object reuse, suppose we were going to encode a series of names. The ASN.1 type for the names would be as follows:

```
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
   givenNameIA5String,
   initial IA5String,
   familyNameIA5String
}
```

The generated C# class would contain public member variables for each of the string objects:

```
public Asn1IA5String givenName;
public Asn1IA5String initial;
public Asn1IA5String familyName;
```

The most efficient way to repopulate these variables within a loop would be simply to assign each of the new strings to be encoded directly to the public *mValue* member variables contained within the *Asn1IA5String* objects (i.e., the *Name* or *Asn1IA5String* objects should not be reconstructed each time).

A code snippet showing how this could be done is as follows:

```
// Step 1: Create Name and Asn1BerEncodeBuffer objects for use in
// the loop..
```

```
Name name = new Name (“”, “”, “”); // creates empty string objects
Asn1BerEncodeBuffer encodeBuffer = new Asn1BerEncodeBuffer ();

for (;;) {

   // logic here to read name components from a DB or other medium

   ...

   // populate string variables (assume string1, 2, and 3 are string
   // variables read from DB above)..

   name.givenName.mValue = string1;
   name.initial.mValue = string2;
   name.familyName.mValue = string3;

   // encode

   try {
      len = name.Encode (encodeBuffer, true);

      // do something with the encoded message component

      ...

      // reset encode buffer for next pass

      encodeBuffer.Reset ();
   }
   catch (Asn1Exception e) {
      // handle error ..
   }
}
```

Stream-Oriented Indefinite Length Encode Methods

BER messages can be encoded directly to an output stream such as a file, network or memory stream. The ASN1C compiler has the **–stream** option to generate encode functions of this type. For each ASN.1 production defined in the ASN.1 source file, a C# encode method may be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

The basic principles of the generation of the encode methods are the same as for the memory-buffer based BER/DER encode methods described in the preceding section. Stream-oriented BER encoding starts from the beginning of the message until the message is complete. This is sometimes referred to as “forward encoding”. This differs from memorybuffer based BER where encoding that is done from back-to-front. Indefinite lengths are used for all constructed elements in the message. Also, there is no permanent buffer for stream-oriented encoding, all octets are written directly to the output stream.

Generated C# Method Format and Calling Parameters

The signature for a C# BER stream-oriented encode method is as follows:

```
public override void Encode (Asn1BerOutputStream outs, bool explicitTagging)
```

The *outs* argument is a reference of an *Asn1BerOutputStream* object that describes the output stream into which a message is being encoded. This must be created and initialized before calling any encode method. See the description of this class in the *C# Run-Time Classes* section for details on how this class is used.

The *explicitTagging* argument specifies whether or not an explicit tag should be applied to the encoded contents. The average user will almost always want to set this argument to true. The only time it would not be set to *true* is if a user wanted to just encode a contents field with no tag. This argument is used primarily by the compiler when generating internal calls to properly handle implicit and explicit tagging.

Unlike the C/C++ version, a negative value is never returned form encode methods to indicate an encoding failure. That is handled by the exception mechanism. All ASN1C C# exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown. If an I/O error occurs then the *System.Exception* is thrown.

Procedure for Calling C# BER Stream-Oriented Encode Methods

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an output stream object into which the value will be encoded

2. Invoke the encode method

3. Close the output stream.

The first step is the creation of an output stream object. There are two forms of the constructor: a constructor with one parameter (*System.IO.Stream* reference) and one that allows specification of an internal buffer size. A larger internal buffer size generally provides better performance at the expense of increased memory consumption. The first constructor sets the value to a reasonable mid-range value.

The second step is the invocation of the Encode() method. The calling arguments were described earlier. As per the C# standard, this method must be invoked from within a try/catch block to catch the possible *Asn1Exception* and *System.Exception*, which may be thrown. Alternatively, the method from which the encode method is called can declare that it throws *Asn1Exception* and *System.Exception* leaving it to be dealt with at a higher level.

Finally, close the output stream.

A complete example showing how to invoke a stream-based encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

Asn1BerOutputStream outs = null;

try {
   // Step 1: Create an output stream object. This object uses the
   // default size increment for buffer expansion..

   outs = new Asn1BerOutputStream ( new System.IO.FileStream(
      filename, System.IO.FileMode.Create));

   // Step 2: Invoke the encode method. Note that it must be done
   // from within a try/catch block..
```

```
      personnelRecord.Encode (outs, true);

      if (trace) {
         System.Console.Out.WriteLine ("Encoding was successful");
         System.Console.Out.WriteLine ("Hex dump of encoded record:");
         encodeBuffer.HexDump ();
         System.Console.Out.WriteLine ("Binary dump:");
         encodeBuffer.BinDump ();
      }
   }
   catch (Exception e) {
      System.Console.Out.WriteLine (e.Message);
      Asn1Util.WriteStackTrace(e, Console.Error);
      return;
   }
   finally {

      // Step 3: Close the output stream, if opened

      try {
         if (outs != null)
             outs.Close ();
      }
      catch (Exception e) {}
   }
```

If you compare this example with the BER encoding example in Figure 1, you will see the encoding procedure is almost identical. This makes it very easy to switch encoding methods should the need arise. All you need to do is change *Asn1BerEncodeBuffer* to *Asn1BerOutputStream* and remove the explicit code that writes the messages into the stream. Also closing of the stream should be added.

Generated BER/DER/CER Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a C# decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the C# object.

As was the case for encode methods, a decode method is only generated if it is required to alter the default method in the base class. The C# model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types. These default implementations include support for handling the universal tags associated with the types as defined in the X.680 standard.

Generated C# Method Format and Calling Parameters

The signature for a C# BER decode method is as follows:

```
public override void Decode (Asn1BerDecodeBuffer buffer, bool explicitTagging,
                             int implicitLength)
```

The *buffer* argument is a reference of an *Asn1BerDecodeBuffer* object that describes the message that is being decoded. This must be created and initialized before calling any decode method. See the description of this class in the *C# Run-Time Classes* section for details on how this class is used.

The *explicitTagging* and *implicitLength* arguments specify whether or not an explicit tag should be parsed from the encoded contents. The average user will almost always want to set *explicitTagging* to true and *implicitLength* to zero. The only time these arguments would not be set this way is if a user wanted to directly decode contents with no tag/length information. These arguments are used primarily by the compiler when generating internal calls to properly handle implicit and explicit tagging.

The decode method returns no result. Unlike the C/C++ version, a negative status value is not returned to indicate a failure. That is handled by the exception mechanism. All ASN1C C# exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various ASN.1 exceptions that can be thrown. The *System.Exception* that can be thrown is in the *Read* method within the decode buffer base class, for any IO errors. This method attempts to read data from an input stream using the methods in the *System.IO.Stream*.

Procedure for Calling C# BER Decode Methods

The general procedure to decode an ASN.1 BER message involves the following three steps:

1. Create a decode message buffer object to describe the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode message buffer object. The *Asn1BerDecodeBuffer* object contains constructors that can either accept a message as a byte array or as an I/O input stream. The input stream option makes it possible to decode messages directly from other mediums other than a memory buffer (for example, a message can be decoded directly from a file).

The *Asn1BerDecodeBuffer* object contains a method called PeekTag that can be used to determine the outer-level tag on a message. This can be used to determine the type of message received in applications that must deal with multiple message types.

The final step is to process the data. All data is contained within public member variables so access is quite easy. And of course C# has the distinct advantage of not requiring any clean-up once you are done with the data. The garbage collector will collect the unused memory when it is no longer referenced.

A complete example showing how to invoke a decode method is as follows:

```
try {

   // Step 1: create a decode message buffer object to describe the
   // message to be decoded. This example will use a file input
   // stream to decode a message directly from a binary file..

   // Create an input file stream object

   System.IO.FileStream ins = new System.IO.FileStream(
      filename, System.IO.FileMode.Open, System.IO.FileAccess.Read);

   // Create a decode buffer object

   Asn1BerDecodeBuffer decodeBuffer = new Asn1BerDecodeBuffer (ins);

   // Step 2: create an object of the generated type and invoke the
   // decode method..

   PersonnelRecord personnelRecord = new PersonnelRecord ();
   personnelRecord.Decode (decodeBuffer);

   // Step 3: process the data

   if (trace) {
      System.Console.Out.WriteLine ("Decode was successful");
      personnelRecord.Print ("personnelRecord");
   }
}
catch (Exception e) {
   System.Console.Out.WriteLine (e.Message);
   Asn1Util.WriteStackTrace(e, Console.Error);
   return;
}
```

Reuse of C# Decoding Objects

The sample above showed the BER decoding of a single message. In a typical application, a loop would be involved to decode a series of messages. While it would be possible to use the code shown above in a loop, it would not be the most efficient way to decode the messages. Objects should be reused where possible to avoid the overhead of excessive memory allocations and garbage collection.

A single decode buffer object can be used to process a stream of messages. If the decode message buffer is created using an input stream object that contains a series of messages (for example, a file containing multiple records or a

communications device), all that needs to be done is the continuous invocation of the BER decode method for the given message type.

Nothing special needs to be done to reuse the generated type object for decoding. The decoder will automatically all the internal *Init()* method before decoding to make sure all items are reset to their starting state.

In the example above, all that would need to be done to decode a series of personnel records is the inclusion of a loop after the PersonnelRecord object was created in step 2:

```
for (;;) {
   personnelRecord.Decode (decodeBuffer);

   if (trace) {
      System.Console.Out.WriteLine ("Decode was successful");
      personnelRecord.Print ("personnelRecord");
   }
}
```

Generated PER Encode Methods

The generation of methods to encode data in accordance with the Packed Encoding Rules (PER) is similar to how methods were generated in the BER/DER case discussed previously. For each ASN.1 production defined in the ASN.1 source file, a C# encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the encoding of the base class method. The C# model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated C# class inherits the *Asn1Integer* base class and the default encode method within this class is sufficient to encode a value of the generated type.

In the case of BER/DER, a custom encode method is generated if a) the type is tagged, or b) it contains a testable constraint. In the case of PER, only the latter condition will cause a custom method to be generated. The reason is because PER basically ignores the tags on tagged types and they therefore have no effect on the final encoded message component.

For example, the following declaration will cause a custom encode method to be generated because the value range constraint is a PER-visible constraint that will alter the encoding:

```
X ::= INTEGER (1..255)
```

In this case, special logic is necessary to apply the value range constraint. Some types will always cause encode methods to be generated. At the primitive level, this is true for the ENUMERATED type. This type will always contain a custom set of enumerated values.

All constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will cause encode methods to be added to the generated classes.

Generated C# Method Format and Calling Parameters

The signature for a C# PER encode method is as follows:

```
public override void Encode (Asn1PerEncodeBuffer buffer)
```

The *buffer* argument is a reference of an *Asn1PerEncodeBuffer* object that describes the buffer into which a message is to be encoded. This must be created and initialized before calling any encode method. See the description of this class in the *C# Run-Time Classes* section for details on how this class is used.

The PER encode methods do not return a value. This is different than the C/C++ version that returns a negative status value to indicate an encoding failure. For C#, errors are reported via the exception mechanism. All ASN1C C# exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Procedure for Calling C# PER Encode Methods

The C# class variables corresponding to each of the ASN.1 types and method of population are the same as they were in the BER encoding case. See the section on BER encoding for instructions on how to populate the variables prior to encoding.

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded

2. Invoke the encode method

3. Invoke encode message buffer methods to access the encoded message component

The first step is the creation of an encode message buffer object. For PER encoding, this is an object of the *Asn1PerEncodeBuffer* class. The following constructors are available for creating a PER encode buffer object:

```
public Asn1PerEncodeBuffer (bool aligned);

public Asn1PerEncodeBuffer (bool aligned, int sizeIncrement);
```

The first argument indicates whether PER aligned or unaligned encoding should be done. The second form of the constructor contains a size increment argument. This argument will determine how often the buffer will need to be resized to hold large messages. If you know that you will be encoding large messages, then this object should be constructed with a large value for the increment. If you know that you will be encoding small messages in a constrained environment, then this value can be set very low. The default constructor sets the value to a reasonable mid-range value (see SIZE_INCREMENT in *Asn1EncodeBuffer.cs*, as of this writing the value was set to 1024).

The second step is the invocation of the encode method. The calling arguments were described earlier. As per the C# standard, this method must be invoked from within a try/catch block to catch the possible exceptions that may be thrown. Alternatively, the method from which the encode method is called can declare that it throws an *Asn1Exception* leaving it to be dealt with at a higher level.

Finally, encode buffer methods can be called to access the encoded message component. The C# API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object provides a method called *GetInputStream* that returns a byte array input stream representing the message component. This is the preferred way to access the encoded component.

In addition to *GetInputStream* there is a *MsgCopy* property that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. Another option that is only available when doing PER encoding is the *Buffer* property. This returns a reference to the actual message buffer into which the message was encoded. Since a PER message is encoded front-to-back (unlike the back-to-front used in BER/DER encoding), the buffer reference returned will point to the start of the encoded message. The *MsgByteCnt* property can then be used to get the message length in bytes or the *MsgBitCnt* property can be called to get the length in bits.

The encode buffer class also contains other methods for operating directly on the encoded component (for example, the write method can be used to write it to a file or other medium). And of course, one could derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1PerEncodeBuffer* class in the runtime section for a full description of the available methods.

A complete example showing how to invoke a PER encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

// Step 1: Create a message buffer object. This object uses the
```

```
// default size increment for buffer expansion..

Asn1PerEncodeBuffer encodeBuffer = new Asn1PerEncodeBuffer();

// Step 2: Invoke the encode method. Note that it must be done
// from within a try/catch block..

try {
   personnelRecord.Encode (encodeBuffer);

   if (trace) {
      System.Console.Out.WriteLine ("Encoding was successful");
      System.Console.Out.WriteLine ("Hex dump of encoded record:");
      encodeBuffer.HexDump ();
      System.Console.Out.WriteLine ("Binary dump:");
      encodeBuffer.BinDump (“personnelRecord”);
   }

   // Step 3: Access the encoded message component. In this
   // case, we use methods in the class to write the component
   // to a file and output a formatted dump to the message.dmp
   // file..

   // Write the encoded record to a file
   encodeBuffer.Write(new System.IO.FileStream(
      filename, System.IO.FileMode.Create));

   // Generate a dump file for comparisons
   System.IO.StreamWriter messagedmp =
      new System.IO.StreamWriter( new System.IO.FileStream(
         "message.dmp", System.IO.FileMode.Create));
   messagedmp.AutoFlush = true;
   encodeBuffer.HexDump(messagedmp);

   // We can also directly access the buffer as follows:

   byte[] buffer = encodeBuffer.Buffer;
   int msglen = encodeBuffer.MsgByteCnt;
}
catch (Exception e) {
   System.Console.Out.WriteLine (e.Message);
   Asn1Util.WriteStackTrace(e, Console.Error);
   return;
}
```

If you compare this example with the BER encoding example in Figure 1, you will see the encoding procedure is almost identical. This makes it very easy to switch encoding methods should the need arise. All you need to do is change *Asn1BerEncodeBuffer* to *Asn1PerEncodeBuffer* and remove the explicit argument from the encode method call.

Reuse of C# Encoding Objects

The concept of reusing C# objects for PER encoding is the same as was described previously for BER encoding. Basically, all that needs to be done is the creation of a single PER encode buffer object and an object corresponding

to the ASN.1 data type to be encoded outside of the processing loop. These objects can then be reused to encode each instance of the messages to be sent. After each message is encoded, the PER buffer must be reset for the next message by using the Reset method. See the section on reuse of objects in the BER encoding section for a more thorough discussion and sample code on using this capability.

Generated PER Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a C# decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the C# object.

As was the case for encode methods, a decode method is only generated if it is required to alter the default method in the base class. The C# model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For primitive types, a custom PER decode method is only generated if one or more of the following is true:

1. The type contains a PER-visible constraint

2. The generation of event handlers was specified

The exception to this rule is the ENUMERATED primitive type (or likewise, INTEGER type with a named number list) that will always cause a decode method to be generated.

Constructed types will always cause custom PER decode methods to be generated.

Generated C# Method Format and Calling Parameters

The signature for a C# PER decode method is as follows:\

```
public override void Decode (Asn1PerDecodeBuffer buffer)
```

The *buffer* argument is a reference of an *Asn1PerDecodeBuffer* object that describes the message that is being decoded. This must be created and initialized before calling any decode method. See the description of this class in the *C# Run-Time Classes* section for details on how this class is used.

The decode method returns no result. Unlike the C/C++ version, a negative status value is not returned to indicate a failure. That is handled by the exception mechanism. All ASN1C C# exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various ASN.1 exceptions that can be thrown. The *System.Exception* that can be thrown is in the *read* method within the decode buffer base class. This method attempts to read data from an input stream using the methods in the *System.IO.Stream* class.

Procedure for Calling C# PER Decode Methods

The general procedure to decode an ASN.1 PER message involves the following three steps:

1. Create a decode message buffer object to describe the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode message buffer object. The *Asn1PerDecodeBuffer* object contains constructors that can either accept a message as a byte array or as an I/O input stream. The input stream option makes it possible to decode messages directly from other mediums other than a memory buffer (for example, a message can be decoded directly from a file or a socket).

Unlike BER or DER, no mechanism exists in PER to peek at an outer level tag or identifier to identify the message type. This type must be known beforehand. Most protocols that employ PER have a specific outer level type know

as a “Protocol Data Unit” (PDU) that encompasses all of the different message types that might be received. This is typically a CHOICE construct with each option representing a different type of message.

The generated decode method for the PDU is invoked to decode the message. The calling arguments were described earlier. As per the C# standard, this method must be invoked from within a try/catch block to catch the possible exceptions that may be thrown. Alternatively, the method from which the decode method is called can declare that it throws the exceptions leaving them to be dealt with at a higher level.

The final step is to process the data. All data is contained within public member variables so access is quite easy. All of the primitive data type classes contain a public member variable called *mValue* that contains decoded data. This can be accessed in nested structures by prefixing *mValue* with each of the element names from the top down. For example, the *givenName* element in the *Name* type shown earlier would be accessed as follows: *name.givenName.mValue* (this assumes an instance of the *Name* class was created using the variable name “*name*”).

A complete example showing how to invoke a decode method is as follows:

```
try {

   // Step 1: create a decode message buffer object to describe the
   // message to be decoded. This example will use a file input
   // stream to decode a message directly from a binary file..

   // Create an input file stream object

   System.IO.FileStream ins = new System.IO.FileStream(
      filename, System.IO.FileMode.Open, System.IO.FileAccess.Read);

   // Create a decode buffer object

   Asn1PerDecodeBuffer decodeBuffer = new Asn1PerDecodeBuffer (ins);

   // Step 2: create an object of the generated type and invoke the
   // decode method..

   PersonnelRecord personnelRecord = new PersonnelRecord ();
   personnelRecord.Decode (decodeBuffer);

   // Step 3: process the data

   if (trace) {
      System.Console.Out.WriteLine ("Decode was successful");
      personnelRecord.Print ("personnelRecord");
   }
}
catch (Exception e) {
   System.Console.Out.WriteLine (e.Message);
   Asn1Util.WriteStackTrace(e, Console.Error);
   return;
}
```

Reuse of C# Decoding Objects

C# objects can be reused for decoding PER messages in the same way they were for BER messages. The decode buffer and message type objects are created outside of the main decoding loop. Then in the main loop these objects are

reused to process each input message. Data must be saved from the message type object after each iteration because the contents of the object will be overwritten on each consecutive loop iteration. Nothing special needs to be done at the bottom of the loop to ready the decoder for the next message. All necessary initialization will be handled internally.

Generated XER / XML Encode Methods

The generation of methods to encode data in accordance with the XML Encoding Rules (XER) or for XSD-compliant XML is similar to how methods were generated in the BER/DER and PER cases discussed previously. For each ASN.1 production defined in the ASN.1 source file, a C# encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the encoding of the base class method. The C# model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated C# class inherits the Asn1Integer base class and the default encode method within this class is sufficient to encode a value of the generated type.

In the case of XER or XML, a custom encode method is only generated if:

1. The ASN.1 type is constructed (SEQUENCE, SET, SEQUENCE OF, SET OF, or CHOICE).
2. The ASN.1 type contains a testable constraint (for example, INTEGER (1..100))
3. The ASN.1 type is enumerated. This includes an INTEGER type with named numbers, a BIT STRING with named bit constants, or the ENUMERATED built-in type.

Generated C# Method Format and Calling Parameters

The signature for a C# XER encode method is as follows:

```
public override void Encode (Asn1XerEncoder buffer, string elemName)
```

The signature for a C# XML encode method is similar:

```
public override void Encode (Asn1XmlEncoder buffer, string elemName)
```

The *buffer* argument is a reference to an *Asn1XerEncoder* or *Asn1XmlEncoder* derived object that describes the buffer or output stream into which a message is to be encoded. *Asn1XerEncoder* is a base interface for the *Asn1XerEncodeBuffer* and *Asn1XerOutputStream* classes. Similarly, *Asn1XmlEncoder* is an interface to a pure XML version of these base classes. There is no difference which encode method is used: output stream or message buffer. The generated logic is the same, the difference is only in the first parameter of the encode method. This must be created and initialized before calling any encode method. See the description of this class in the *C# Run-TimeReference Manual* for details on how this class is used.

The elemName argument is a reference to a string containing the element name text. This text is used to form the standard XML angle-bracketed wrapper that is applied to each element in a message. Note the name passed must not contain the angle-brackets (i.e. the < > characters). These will be added by the encode method.

The *elemName* can be passed in different ways to control how the name is applied. The normal way is to pass a name that is applied as the element name of the element. If null is passed, then the default element name for the referenced

ASN.1 built-in type is used. For example, <BOOLEAN> is the default element name for the ASN.1 BOOLEAN type. The complete list of default element names can be found in the X.693 standard. If an empty string is passed (i.e. “”), this tells the encode method to omit the element name string all together and just encode the value (this is similar to implicit tagging in the BER case).

The XER or XML encode methods do not return a value. This is different than the C/C++ version that returns a negative status value to indicate an encoding failure. For C#, errors are reported via the exception mechanism. All ASN1C C# exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown. If I/O error occurs then the *System.Exception* is thrown.

Procedure for Calling C# XER Encode Methods

The C# class variables corresponding to each of the ASN.1 types and method of population are the same as they were in the BER encoding case. See the section *Populating Generated Variables for Encoding* for instructions on how to populate the variables prior to encoding.

Once an object’s member variables have been populated, the object’s encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer or output stream object into which the value will be encoded

2. Invoke encode methods. These include the *EncodeStartDocument* and *EncodeEndDocument* methods from the *Asn1XerEncodeBuffer* class and the encode method from the ASN1C generated class.

3. If the encode message buffer is used: invoke encode message buffer methods to access the encoded message component. If the output stream is used: close the stream.

The first step is the creation of an encode message buffer object. For XER encoding, this is an object of the Asn1XerEncodeBuffer class. The following constructors are available for creating an XER encode buffer object:

```
public Asn1XerEncodeBuffer ();

public Asn1XerEncodeBuffer (bool canonical, int sizeIncrement);
```

The default constructor sets all internal buffer control variables to default values. Canonical XER is set to false and size increment is set to 1024. The other forms of the constructor allow these variables to be changed. Canonical XER specifies that the canonical form of XER encoding (CXER as specified in X.693) should be used. Size increment specifies the amount by which the dynamic encode buffer should be expanded when it fills up. This should be set lower for small, memory-constrained environments and higher if large messages are being encoded.

If the output stream method is used then the first step is the creation of an output stream. For XER encoding, this is an object of the Asn1XerOutputStream class. The following constructors are available for creating an XER encode buffer object:

```
public Asn1XerOutputStream (OutputStream os);

public Asn1XerOutputStream (OutputStream os, bool canonical, int bufSize);
```

The first constructor creates a buffered XER output stream with default size of an internal buffer. Canonical XER is set to false. The other form of the constructor allows these variables to be changed. Canonical XER specifies that the canonical form of XER encoding (CXER as specified in X.693) should be used. The buffer size argument specifies the size of the internal buffer of the stream. Larger buffer sizes typically provide better performance at the expense of increased memory consumption.

Similar classes exist for XML encode buffer and streams:

```
public Asn1XmlEncodeBuffer ()
public Asn1XmlEncodeBuffer (int sizeIncrement)
public Asn1XmlOutputStream (OutputStream os)
public Asn1XmlOutputStream (OutputStream os, int bufSize)
```

The main difference is the XML classes to not have a canonical XML option; therefore, there is not *cxer* or *canonical* boolean argument.

The second step is the invocation of the encode methods. The calling arguments were described earlier. As per the C# standard, this method must be invoked from within a try/catch block to catch the possible *Asn1Exception* or *System.Exception* that may be thrown. Alternatively, the method from which the encode method is called can declare that it throws *Asn1Exception* and *System.Exception* leaving it to be dealt with at a higher level.

Finally, if a message buffer is used, encode buffer methods can be called to access the encoded message component. The C# API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object provides a method called *GetInputStream* that returns a byte array input stream representing the message component. This is the preferred way to access the encoded component.

In addition to *GetInputStream*, there is a *MsgCopy* property that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. Another option that is available when doing XER encoding is the *Buffer* property. This returns a reference to the actual message buffer into which the message was encoded. Since an XER message is encoded front-to-back (unlike the back-to-front used in BER/DER encoding), the buffer reference returned will point to the start of the encoded message. The *MsgLength* property can then be used to get the message length (in bytes). Note that the byte count may not correspond to the actual character count as UTF-8 encoding is used and some characters may be multiple bytes in length.

If an output stream is used, the stream should be closed when encoding is complete to ensure all buffered data is flushed to the output device.

The *Asn1XerEncodeBuffer* encode *buffer* class also contains other methods for operating directly on the encoded component (for example, the write method can be used to write it to a file or other medium). A user could also derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1XerEncodeBuffer* class in the run-time section for a full description of the available methods.

A complete example showing how to invoke an XER encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

// Step 1: Create a message buffer object. This object uses
// standard XER (non-canonical) and the default size increment
// for buffer expansion..

Asn1XerEncodeBuffer encodeBuffer = new Asn1XerEncodeBuffer();

// Step 2: Invoke the encode methods. These include
// encodeStartDocument to encode the XML document header,
// the generated C# encode method to encode the document body,
// and the encodeEndDocument method to complete the message.
// Note that these methods must be invoked from within a
// try/catch block..

try {
   encodeBuffer.EncodeStartDocument ();

   personnelRecord.Encode (encodeBuffer, null);
```

```
      encodeBuffer.EncodeEndDocument ();

      if (trace) {
         System.Console.Out.WriteLine ("Encoding was successful");
         encodeBuffer.Write (System.Console.OpenStandardOutput());
      }

      // Step 3: Access the encoded message component. In this
      // case, we use methods in the class to write the encoded
      // XML document to a file..

      encodeBuffer.Write(new System.IO.FileStream(filename,
         System.IO.FileMode.Create));

      // We can also directly access the buffer as follows:

      byte[] buffer = encodeBuffer.Buffer;
      int msglen = encodeBuffer.MsgByteCnt;
   }
   catch (Exception e) {
      System.Console.Out.WriteLine (e.Message);
      Asn1Util.WriteStackTrace(e, Console.Error);
      return;
   }
```

An example showing stream-based encoding is as follows:

```
   // Note: personnelRecord object was previously populated with data

   Asn1XerOutputSteram outs = null;

   try {
      // Step 1: Create an output stream object. This object
      // uses standard XER (non-canonical) and the default
      // internal buffer's size.

      outs = new Asn1XerOutputStream(new System.IO.FileStream(
         filename, System.IO.FileMode.Create));

      // Step 2: Invoke the encode methods. These include
      // encodeStartDocument to encode the XML document header,
      // the generated C# encode method to encode the document body,
      // and the encodeEndDocument method to complete the message.
      // Note that these methods must be invoked from within a
      // try/catch block..

      outs.EncodeStartDocument ();

      personnelRecord.Encode (outs, null);

      outs.EncodeEndDocument ();

      if (trace) {
```

```
            System.Console.Out.WriteLine ("Encoding was successful");
            encodeBuffer.Write (System.Console.OpenStandardOutput());
        }
    }
    catch (Exception e) {
        System.Console.Out.WriteLine (e.Message);
        Asn1Util.WriteStackTrace(e, Console.Error);
        return;
    }
    finally {
        // Step 3: Close the stream.

        try {
            if (outs != null)
                outs.Close ();
        }
        catch (Exception e) {}
    }
```

If you compare these examples with the other encoding examples, you will see the procedures are similar. This makes it very easy to switch encoding methods should the need arise.

In the case of XML encode, the procedure is very similar. The only difference is that it is not necessary to call the *encodeStartDocument* and *encodeEndDocument* methods. The are built into the generated *encode* method for PDU data types.

The resulting XML document from running the program above is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonnelRecord>
   <name>
      <givenName>John</givenName>
      <initial>P</initial>
      <familyName>Smith</familyName>
   </name>
   <number>51</number>
   <title>Director</title>
   <dateOfHire>19710917</dateOfHire>
   <nameOfSpouse>
      <givenName>Mary</givenName>
      <initial>T</initial>
      <familyName>Smith</familyName>
   </nameOfSpouse>
   <children>
      <ChildInformation>
         <name>
            <givenName>Ralph</givenName>
            <initial>T</initial>
            <familyName>Smith</familyName>
         </name>
         <dateOfBirth>19571111</dateOfBirth>
      </ChildInformation>
      <ChildInformation>
         <name>
```

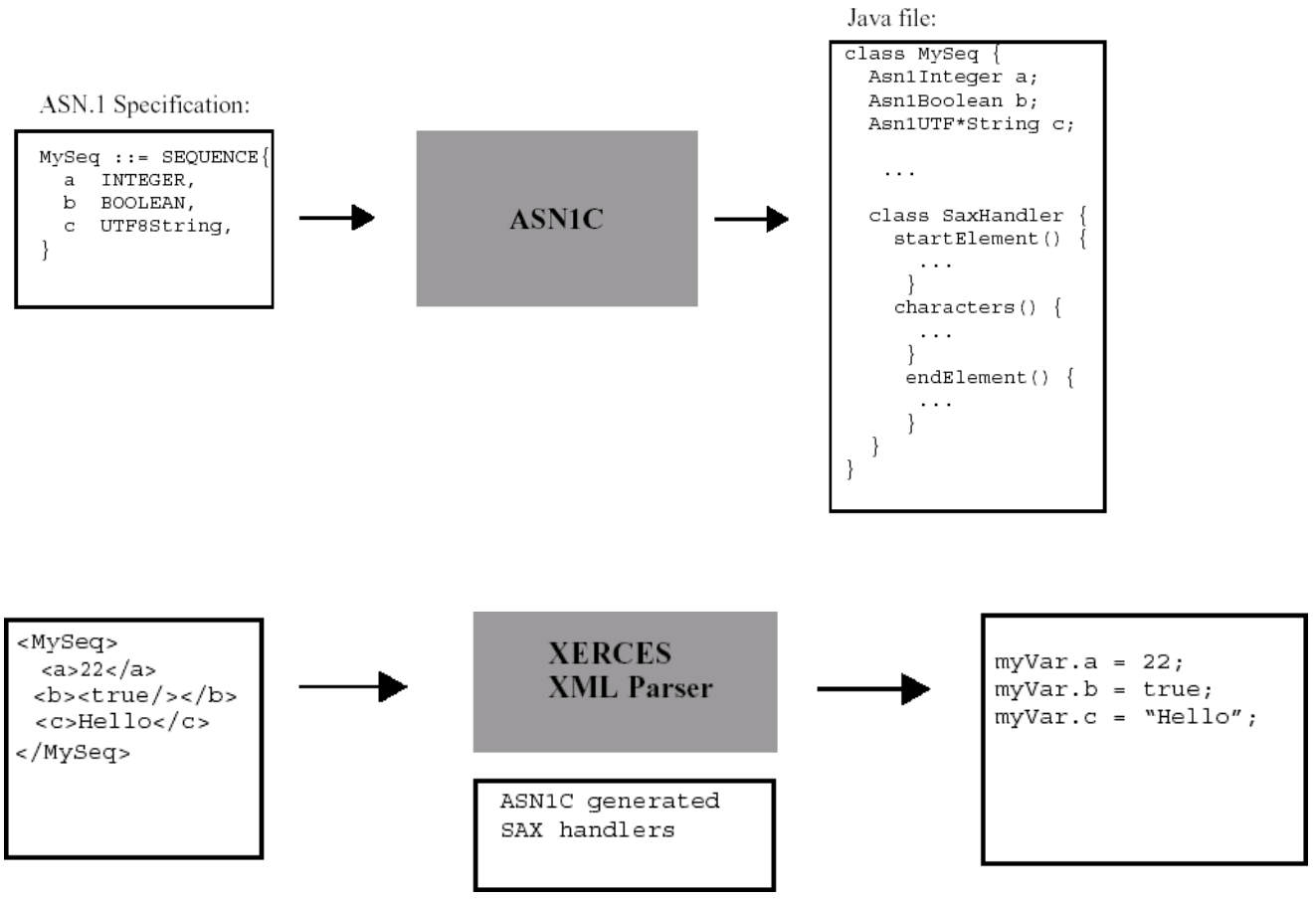
```
            <givenName>Susan</givenName>
            <initial>B</initial>
            <familyName>Jones</familyName>
          </name>
          <dateOfBirth>19590717</dateOfBirth>
        </ChildInformation>
    </children>
</PersonnelRecord>
```

Generated XER / XML Decode Methods

The code generated to decode XML messages is different than that of the other encoding rules. This is because off-theshelf XML parser software is used to parse the XML documents to be decoded. This software contains a common interface known as the *Simple API for XML (or SAX)* that is a de-facto standard that is supported by most parsers. ASN1C generates an implementation of the content handler interface defined by this standard. This implementation receives the parsed XML data and uses it to populate the structures generated by the compiler.

The default XML parser used for C# is the MSXML parser that is part of the Microsoft .NET framework. As mentioned, since SAX is a de-facto standard, it should be a relatively straightforward process to use the generated handlers with any other parser.

A diagram showing the components used in the XML decode process is as follows:



ASN1C generates code to implement the following methods defined in the SAX content handler interface:

```
StartElement

Characters

EndElement
```

The interface defines other methods that can be implemented as well, but these are sufficient to decode XER encoded data. These methods are added to an inner SAX handler class generated for each ASN.1 production.

The procedure to invoke the generated decode method is as follows:

1. Instantiate an *XmlSaxParser* object.

2. Instantiate a generated C# <ProdName> object to hold the decoded message data.

3. Invoke the <ProdName> object decode method passing the reader created in step 1 and the URI of the XML document to be parsed. This method initiates and invokes the XML parser's *parse* method to parse the document. This, in turn, invokes the generated SAX handler methods.

4. Methods within the <ProdName> object can now be used to access the decoded data. The member variables that were declared to be public can be accessed directly.

5. Error handling is accomplished using a try-catch block to catch SAX exceptions.

A program fragment that could be used to decode an employee record is as follows:

```
public class Reader {
  public static void main (string args[]) {
     string filename = "employee.xml";

     try {
        // Create an XML reader object

        XmlSaxParser reader = XmlSaxParser.NewInstance();

        // Read and decode the message

        PersonnelRecord personnelRecord = new PersonnelRecord ();
        personnelRecord.Decode (reader, filename);
        if (trace) {
           System.Console.Out.WriteLine ("Decode was successful");
           personnelRecord.Print ("personnelRecord");
        }
     }
     catch (Exception e) {
        System.Console.Out.WriteLine (e.Message);
        Asn1Util.WriteStackTrace(e, Console.Error);
        return;
     }
  }
}
```

Table Constraint Processing

The ASN1C C# code generation capability can generate code to process ASN.1 table constraints as specified in the X.681 and X.682 ASN.1 standards. This code is generated through the use of the **-tables** option. This instructs the compiler to generate additional methods and tables to allow multi-level message types specified using table constraints to be encoded or decoded with a single method call.

Special code is generated for the CLASS, Information Object, and Information Object Set items to create the table necessary to for table constraint processing. Then additional encode and decode methods are generated that use these tables to branch to the multiple message levels.

CLASS Specification

NOTE: Class code generation is done only when -tables is specified.

This additional code is generated to support the processing required to verify table constraints, which is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

The C# class generated to model an ASN.1 class contains member variables for each of the fields within the class. To create an instance of this class, an information object is required to populate these variables with the values defined in the ASN.1 information object specification.

C# code will be generated for each ASN.1 CLASS definition in a separate C# source file containing a C# class corresponding to the ASN.1 CLASS definition. The name of the source file and class is of the following format:

```
<ClassName>.cs
```

In this definition, <*ClassName*> would be replaced with the name of the ASN.1 CLASS for which this file is generated.

Data Member Generation

For each of the following ASN.1 CLASS fields, a corresponding member variable is generated in the C# class definition:

For a value field:

```
public <TypeName> <FieldName>;
```

For a type field:

```
public Asn1Type <FieldName>;
```

For an information object field:

```
public <ClassName> <FieldName>;
```

For an information object set field:

```
public <ClassName> <FieldName>;
```

where:

<FieldName> is replaced with the name of the field.

<TypeName> is replaced with the generated runtime C# classname for the ASN.1 Type.

<ClassName> is replaced with the name of the information object class.

For a type field definition, an element with type Asn1Type is generated which is the base class for all types in the C# runtime namespace. A type field can hold a value of any type.

Method and Constructor Generation

Each generated C# class will have two constructors. The first constructor will be the default constructor. This will initialize each member variable value to *null*. The second constructor will accept values for all the data members.

Example

As an example, consider the following ASN.1 class definition :

```
ATTRIBUTE ::= CLASS {
   &Type,
   &id OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX { WITH SYNTAX &Type ID &id }
```

A file named ATTRIBUTE.cs is generated with following definition:

```
public class ATTRIBUTE {
  public Asn1Type Type;
  public Asn1ObjectIdentifier id;

  public ATTRIBUTE() {
     Type = null;
     id = null;
  }

  public ATTRIBUTE(
     Asn1Type Type_,
     Asn1ObjectIdentifier id_
     ) {
     Type = Type_;
     id = id_;
  }
}
```

NOTE: If the ASN.1 type name is same as the ASN.1 class name (ignoring case) in a single module definition, then the ASN.1 class name will be changed to the following:

```
<ClassName>_CLASS
```

In this definition, <ClassName> would be replaced with the name of the ASN.1 CLASS and the literal token "_CLASS" would be appended.

For example:

```
Test DEFINITION ::= BEGIN
```

```
   Attribute ::= INTEGER
   ATTRIBUTE ::= ABSTRACT-SYNTAX
END
```

ASN1C will change the ATTRIBUTE class name to ATTRIBUTE_CLASS to avoid conflicts with the Attribute type.

This automated feature will help users to successfully compile the generated code without having to manually change the name via a configuration file setting.

Additional C# classes are generated to create types for fields within the class definitions as follows:

1. New type assignments are created for TypeField type definitions as follows:

   ```
   _<ClassName>_<FieldName> ::= <Type>
   ```

   Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of the field. Type is the type definition in the ASN.1 CLASS's TypeField.

   This type is used as a defined type in the information object definition for absent values of the TypeField. It is also useful for the user to generate a value for a related OpenType definition in a table constraint.

2. New type assignments are created for ValueField or ValueSetField type definitions if the type is with a constraint definition and/or the type is Sequence / Set / Choice / Sequenceof / SetOf definition.

   ```
   _<ClassName>_<FieldName> ::= <Type>
   ```

   Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of the ValueField or ValueSetField. Type is the type definition in The ASN.1 CLASS's ValueField or ValueSetField. This type will appear as a defined type in the ASN.1 CLASS's ValueField or ValueSetField.

   This new type assignment is used for compiler internal code generation purpose. It is not designed for use by the end user.

3. New value assignments are created for ValueField default value definitions as follows:

   ```
   _<ClassName>_<FieldName>_default <Type> ::= <Value>
   ```

   Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of the ValueField. Value is the default *value* in the ASN.1 CLASS's ValueField and Type is the *type* in the ASN.1 CLASS's ValueField.

   This value is used as a defined value in the information object definition for an absent value of the field. This new value assignment is used for compiler internal code generation purpose. It is not designed for use by the end user.

ABSTRACT-SYNTAX class

The ASN.1 ABSTRACT-SYNTAX class is a useful class definition used to declare the top-level protocol data units (PDU's) defined within a specification. The class is described using the following ASN.1 definition:

```
ABSTRACT-SYNTAX ::= CLASS {
   &id OBJECT IDENTIFIER UNIQUE,
   &Type,
   &property BIT STRING { handles-invalid-encoding(0)} DEFAULT {}
}
WITH SYNTAX {
   &Type IDENTIFIED BY &id [HAS PROPERTY &property]
```

```
}
```

ASN1C is used to create a meta-definition for this structure. The definition will be generated in the file *Asn1AbstractSyntax.cs* (or *Asn1XerAbstractSyntax.cs* for XER), if needed by compiling ASN.1 definitions. An object created from the resulting C# class is populated just like any other compiler-generated structure for working with ASN.1 data.

TYPE-IDENTIFIER class

The ASN.1 TYPE-IDENTIFIER class is a useful class definition for uniquely identifying typed data at runtime. The class is described using the following ASN.1 definition:

```
TYPE-IDENTIFIER ::= CLASS {
   &id OBJECT IDENTIFIER UNIQUE,
   &Type
}
WITH SYNTAX { &Type IDENTIFIED BY &id }
```

The ASN.1 compiler is used to create a meta-definition for this structure. The definition will be generated in the file *Asn1TypeIdentifier.cs* (or *Asn1XerTypeIdentifier.cs* for XER), if needed by compiling ASN.1 definitions. An object created from the resulting C# class is populated just like any other compiler-generated structure for working with ASN.1 data.

Information Object

NOTE: Information Object code generation is only done when the -tables option is selected.

This additional code is generated to support the processing required to verify table constraints, which is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Information Object code will be generated in a C# source file with a special class to hold the values. The name of the source file and class is of the following format:

```
_<ModuleName>Values.cs
```

In this definition, <*ModuleName*> would be replaced with the name of the ASN.1 module in which the values are defined.

For each Information Object defined within a specification, a C# constant is generated which is an instance of the ASN.1 CLASS definition for the object. Each Information Object constant calls the Class constructor with the field value specified in the ASN.1 information object definition.

If the ASN.1 CLASS field is optional and the field value is absent in the Information Object definition, then its corresponding member variable will be initialized to "*null*". If the ASN.1 CLASS field has a default value and its field value is absent in the Information Object, then the generated code for the Information Object will set the Class field's value to the default value.

ASN.1 definition:

```
<name> <ClassName> ::= <InfoObject>
```

Generated C# constants:

```
public static readonly <ClassName> <name> =
   new <ClassName> (<InfoObject values>>);
```

For example, consider the following Information Object declaration for the above ATTRIBUTE class:

```
name ATTRIBUTE ::= {
   WITH SYNTAX VisibleString
   ID { 0 1 1 }
}
```

This would result in the following C# constant being generated:

```
public static readonly ATTRIBUTE name =
   new ATTRIBUTE (
      new Asn1VisibleString(),
      new Asn1ObjectIdentifier(new int[]{0, 1, 1}));
```

NOTE: The following new Type Assignment is created for each TypeField's type definition if the type is one of the following ASN.1 built-in types: Sequence / Set / SequenceOf / SetOf / Choice / Constrained Type / Enumerated Type / NamedInteger Type / NamedBitList Type / ParameterizedType:

```
_<ObjectName>_<FieldName> ::= <Type>
```

Here *ObjectName* is replaced with name of the Object Assignment. If Object is defined in ObjectSet, then *ObjectName* is replaced with the name of the ObjectSet Assignment. *FieldName* is replaced with name of this type field. *Type* is the type definition in Object's typefield.

This type is used as Defined Type in the information object definition for type field. It is also useful for the user to generate value for related OpenType definition in table constraint.

Information Object Set

NOTE: Information Object Set code generation is only done when the -tables option is selected.

This additional code is generated to support the processing required to verify table constraints which is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Information Object code will be generated in a C# source file with a special class to hold the values. The name of the source file and class is of the following format:

```
_<ModuleName>Values.cs
```

In this definition, <*ModuleName*> would be replaced with the name of the ASN.1 module in which the Information Object Sets are defined.

Each Information Object Set specification causes a C# constant to be generated containing an array of Information Object values. Each object in the array is an instance of the equivalent C# class representing the corresponding ASN.1 information object

As of this writing, a static array is used to hold the objects, but this could be changed to something like a linked list or hash.

ASN.1 definition:

```
   <name> <ClassName> ::= { <Information Object1> | <Information Object2> }
```

Generated C# constants:

```
   public static readonly <ClassName> <name> =
      new <ClassName> {<Information Object1>, <Information Object1> };
```

For example, consider the following Information Object Set declaration for above ATTRIBUTE definition:

```
   SupportedAttributes ATTRIBUTE ::= { name | commonName }
```

This would result in the following C# constant being generated:

```
   public static readonly ATTRIBUTE[] SupportedAttributes =
new ATTRIBUTE[] {
_TestValues.name,
_TestValues.commonName
};
```

Generated Information Object Table Structure

Information Objects and Classes are used to define multi-layer protocols in which “holes” are defined within ASN.1 types for passing message components to different layers for processing. These items are also used to define the contents of various messages that are allowed in a particular exchange of messages. The ASN1C compiler extracts the types involved in these message exchanges and generates encoders/decoders for them. The “holes” in the types are accounted for by adding open type holders to the generated structures. These open type holders consist of a byte array for storing information on an encoded message fragment for processing at the next level.

The ASN1C compiler is capable of generating code in one of two forms for information in an object specification:

1. **Simple form:** in this form, references to variable type fields within standard types are simply treated as open types and an open type placeholder is inserted.

2. **Table form:** in this form, all of the classes, objects, and object sets within a specification result in the generation of code for parsing and formatting the information field references within standard type structures.

The second form is selected by specifying the –tables command line option.

To better understand the support in this area, the individual components of Information Object specifications are examined. We begin with the “CLASS” specification that provides a schema for Information Object definitions. A sample class specification is as follows:

```
   OPERATION ::= CLASS {
      &operationCode CHOICE { local INTEGER,
                     global OBJECT IDENTIFIER }
      &ArgumentType,
      &ResultType,
      &Errors ERROR OPTIONAL
   }
```

Users familiar with ASN.1 will recognize this as a simplified definition of the ROSE OPERATION MACRO using the Information Object format. When a class specification such as this is parsed, information on its fields is maintained in memory for later reference. In the simple form of code generation, the class definition itself does not result in the generation of any corresponding C# code. It is only an abstract template that will be used to define new items later

on in the specification. In the table form, a C# container class is generated to hold the Information Object instances of the ASN.1 CLASS.

Fields from within the class can be referenced in standard ASN.1 types. It is these types of references that the compiler is mainly concerned with. These are typically “header” types that are used to add a common header to a variety of other message body types. An example would be the following ASN.1 type definition for a ROSE invoke message header:

```
Invoke ::= SEQUENCE {
   invokeID INTEGER,
   opcode OPERATION.&operationCode,
   argumentOPERATION.&ArgumentType
}
```

This is a very simple case that purposely omits a lot of additional information such as Information Object Set constraints that are typically part of definitions such as this. The reason this information is not present is because we are just interested in showing the items that the compiler is concerned with. We will use this type to demonstrate the simple form of code generation. We will then add table constraints and discuss what changes when the –tables command line options is used.

The opcode field within this definition is an example of a *fixed type* field reference. It is known as this because if you go back to the original class specification, you will see that operationCode is defined to be of a specific type (namely a choice between a local and global value). The generated typedef for this field will contain a reference to the type from the class definition.

The argument field is an example of a *variable type* field. In this case, if you refer back to the class definition, you will see that no type is provided. This means that this field can contain an instance of any encoded type (note: in practice, table constraints can be used with Information Object Sets to limit the message types that can be placed in this field). The generated typedef for this field contains an “open type” (C# *Asn1OpenType* class) reference to hold a previously encoded component to be specified in the final message.

Simple Form Code Generation

In the simple form of information object code generation, the Invoke type above would result in the following C# typedefs being generated:

```
public class Invoke : Asn1Type {
  public Asn1Integer invokeID;
  public OPERATION_operationCode opcode;
  public Asn1OpenType argument;

  ...
}
```

The following would be the procedure to add the Invoke header type to an ASN.1 message body:

1. Encode the body type
2. Get the message bytes and length of the encoded body
3. Plug the bytes into the “data” argument of the open type constructor in the Invoke type variable.
4. Populate the remaining Invoke type fields.
5. Encode the Invoke type to produce the final message.

In this case, the amount of code generated to support the information object references is minimal. The amount of coding required by a user to encode or decode the variable type field elements, however, can be rather large. This

is a trade-off that exists between using the compiler generated table constraints solution (as we will see below) and using the simple form.

Table Form Code Generation

If we now add table constraints to our original type definition, it might look as follows:

```
Invoke ::= SEQUENCE {
   invokeID    INTEGER,
   opcode      OPERATION.&operationCode ({My-ops}),
   argument    OPERATION.&ArgumentType ({My-ops}{@opcode})
}
```

The “{My-ops}” constraint on the opcode element specifies an information object set (not shown) that constrains the element value to one of the values in the object set. The {My-ops}{@opcode} constraint on the argument element goes a step further – it ties the type of the field to the type specified in the row that matches the given opcode value. ASN1C generates an in-memory table for each of the items in the information object sets defined in a specification. In the example above, a table would be generated for the My-ops information object set. The code generated for the type would then use this table to verify that the given items in a structure that reference this table match the constraints. The C# type generated for the SEQUENCE above when –tables is specified would be as follows:

```
public class Invoke : Asn1Type {
  public Asn1Integer invokeID;
  public OPERATION_operationCode opcode;
  public Asn1Type argument;

  ...
}
```

This is almost identical to the type generated in the simple case. The difference is that *ASN1Type* is used instead for the argument element instead of *ASN1OpenType*. This type is defined as the base class for all the generated ASN.1 types. It holds the value to be encoded or decoded. The way a user Would use this to encode a value of this type is as follows:

1. Populate a variable of the type to be used as the argument to the invoke type.

2. Assign it to the argument member variable in the structure above.

3. Populate the remaining Invoke type fields.

4. Encode the Invoke type to produce the final message.

Note that in this case, the intermediate type does not need to be manually encoded by the user. The generated encoder has logic built-in to encode the complete message using the information in the generated tables.

Additional Code Generated for the -tables Option

The following additional code is generated for type definitions when the the -tables command-line option is used. The code generated to support table constraints is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

An *equals()* method will be generated for Sequence, Set, Sequence Of, Set Of or Choice types if required for table constraint processing. This method will be an implementation of the *Asn1Type.equals()* virtual method. These methods are used by the generated code to verify that data in a generated structure to be encoded (or data that has just been decoded) matches the table constraint values.

An additional table constraint check method is also generated for each type that contains table constraints. These functions have the following prototypes:

BER/DER:

```
void CheckTC (bool decode);
```

PER:

```
void CheckTC (bool decode, bool aligned);
```

The *decode* argument is used to decide if this method is to used for encoding or decoding. The *aligned* argument is for PER and specifies whether aligned or unaligned encoding/decoding is in effect.

The purpose of these methods is to verify that the fixed values within the table constraints are what they should be and to encode or decode the open type fields using the encoder or decoder methods from the *Asn1Type* objects assigned to the given table row. Calls to these functions are automatically built into the standard encode or decode functions for the given type. They should be considered hidden functions not for use within an application that uses the API.

The *CheckTC* method will have different logic for relative and simple table constraints. The logic to invoke this method is as follows:

On the encode side:

Relative Table Constraint:

1. The table constraint key is searched in the object set array to find the class object for the data in the populated type variable to be encoded.

2. If the key element value is NOT found and the table constraint object set is extensible, the *CheckTC* method will do no further processing (i.e. a value field match will not be performed). The user will have had to populate the type field using an *Asn1OpenType* object in order for it to be encoded because the generated table contains no information on how to encode the value.

3. If the key element value is found, the method will verify all fixed type values match what is defined in the key row of the object set and will also verify that the type of any variable type fields matches the expected type.

4. If the key element value is not found in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

Simple Table Constraint:

1. The *CheckTC* method will verify that all of the fixed type values match what is defined in the table constraint object set. If the element value does not exist in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

   After the *CheckTC* method call, the normal encode logic is performed.

   For decoding, the logic is reversed:

   The normal decode logic is performed first to populate the standard and open type fields in the generated structure. After that, the *CheckTC* method is invoked to perform the following table constraint checks:

Relative Table Constraint:

1. The table constraint key is searched in the object set array to find the class object for the data in the populated type variable to be encoded.

2. If the key element value is NOT found and the table constraint object set is extensible, the *CheckTC* method will do no further processing (i.e. a value field match will not be performed) and the variable type fields will be stored as open types (i.e. as instances of C# *Asn1OpenType* classes). The user will be responsible for further decoding of the open type value.

3. If the key element value is found, the *CheckTC* method will verify all fixed type values match what is defined in the key row of the object set and will fully decode all type fields according to the key row type and store the resulting decoded type in the *ASN1Type* fields.

4. If the key element value is NOT found in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

Simple Table Constraint:

1. This function will verify all the fixed type values match what is defined in the table constraint object set. If an element value does not exist in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

Populating OpenType Variables for Encoding

When the -tables option is used, open type fields are generated as *Asn1Type* fields. The general procedure to populate the values for these fields is as follows:

1. Check the possible type in the object set from the indexed element value.

2. Populate the value for this type and assign it to the open type member variable.

3. Follow the common encode procedure.

A complete example showing how to assign open type values when table constraint code is generated is as follows:

```
ATTRIBUTE ::= CLASS {
   &Type,
   &id              OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX {
   WITH SYNTAX &Type ID &id }

name ATTRIBUTE ::= {
   WITH SYNTAX     VisibleString
   ID              { 0 1 1 } }

commonName ATTRIBUTE ::= {
   WITH SYNTAX     INTEGER
   ID              { 0 1 2 } }

SupportedAttributes ATTRIBUTE ::= { name | commonName }

Invoke ::= SEQUENCE {
   opcode   ATTRIBUTE.&id   ({SupportedAttributes}),
   argument ATTRIBUTE.&Type ({SupportedAttributes}{@opcode})
}
```

In the above example, the *Invoke* type contains a relative table constraint. Its element *opcode* refers to the ATTRIBUTE class's *id* field and the *argument* element refers to ATTRIBUTE class's *Type* field. The *opcode* element is the index

element into the *{SupportedAttributes}* information object set. The *argument* element is an open type but its type must match that specified at the location in the *{SupportedAttributes}* information object set indexed by *opcode*.

In this example, *opcode* can have only two possible values { 0 1 1 } or { 0 1 2 }. If the *opcode* value is { 0 1 1} then *argument* must be a value of type *VisibleString*. If the *opcode* value is { 0 1 2 } then argument will have an INTEGER value. Any other value of the opcode element will be a violation of the table constraint.

If the *SupportedAttributes* object set is extensible (in this example, it is not), then the *argument* element can be a value of any type. In this case, if the user is using an index element value outside the object set, then the user will have to encode the *argument* element as an *Asn1OpenType*.

The following sample code populates the open type value:

```
// Step 1: populate the “Invoke” type with data
Invoke pdu = new Invoke();
pdu.opcode = new Asn1ObjectIdentifier(new int[]{0, 1, 1});
pdu.argument = new Asn1VisibleString(“objsys”);
// note: opcode value is {0 1 1 }, so argument must be
// Asn1VisibleString type

// note: the rest of the encode method will be same as general
// PER/DER/BER encoding rules

// Step 2: Create a message buffer object.
Asn1PerEncodeBuffer encodeBuffer = new Asn1PerEncodeBuffer();

// Step 3: Invoke the encode method. Note that it must be done
// from within a try/catch block..
try {
   pdu.Encode (encodeBuffer);
   if (trace) {
      System.Console.Out.WriteLine ("Encoding was successful");
      System.Console.Out.WriteLine ("Hex dump of encoded record:");
      encodeBuffer.HexDump ();
      System.Console.Out.WriteLine ("Binary dump:");
      encodeBuffer.BinDump (“Invoke”);
   }

   // Step 3: Access the encoded message component. In this
   // case, we use methods in the class to write the component
   // to a file and output a formatted dump to the message.dmp
   // file..

   // Write the encoded record to a file
   encodeBuffer.Write(new System.IO.FileStream(
      filename, System.IO.FileMode.Create));

   // Generate a dump file for comparisons
   System.IO.StreamWriter messagedmp =
      new System.IO.StreamWriter( new System.IO.FileStream(
         "message.dmp", System.IO.FileMode.Create));
   messagedmp.AutoFlush = true;
   encodeBuffer.HexDump(messagedmp);

   // We can also directly access the buffer as follows:
```

```
      byte[] buffer = encodeBuffer.Buffer;
      int msglen = encodeBuffer.MsgByteCnt;
   }
   catch (Exception e) {
      System.Console.Out.WriteLine (e.Message);
      Asn1Util.WriteStackTrace(e, Console.Error);
      return;
   }
```

The important thing to note is that not much changes from the normal procedure. The only significant difference is that now the *argument* field can be directly populated with an instance of its target type. Without table constraint checking logic, this value would have to have been first encoded and then placed in an *Asn1OpenType* container object.

Decoding Types with Table Constraints

The general procedure to decode an ASN.1 message with table constraint is the same as without table constraints. The only difference is that after decoding, variable type fields will be replaced with instanced of the actual types they are specified to contain in the associated object set instead of with generic *Asn1OpenType* fields.

Generated Print Methods

The –print option causes print methods to be generated. These functions can be used to print the contents of variables of generated types. A print method is generated in each of the generated C# source files.

Generated C# Method Format and Calling Parameters

There are two signatures for generated print methods which are as follows:

The first signature generated allows printing to any output stream:

```
public override void Print (System.IO.StreamWriter outs,
                            string varName,
                            int level)
```

The *outs* argument specifies a *System.IO.StreamWriter* object to which the output will be written. The C# class *System.IO.Stream* may be populated with *System.Console.Out* to write directly to standard output.

The *varName* argument is used to specify the top-level variable name of the item being printed. Normally, this would be set to the same name as the variable declared in your program that holds the object being printed. For example, if you declared a variable called *personnelRecord* to hold a *PersonnelRecord* object, the *varName* object would be set to “personnelRecord”.

The *level* argument is used to specify the indentation level for printing nested types. The user would always want to set this to zero at the outer-level.

The second signature exists in the *Asn1Type* base class. This provides a simplified interface for printing to standard output. In this case, the *outs* and *level* arguments are omitted:

```
public override void Print (string varName)
```

The *varName* argument is same as in the first case above.

For example, the call to print the personnelRecord from the previous examples would be as follows:

```
personnelRecord.Print ("personnelRecord");
```

The output would be formatted as follows:

```
personnelRecord {
   name {
      givenName = 'John'
      initial = 'P'
      familyName = 'Smith'
   }
   number = 51
   title = 'Director'
   dateOfHire = '19710917'
   nameOfSpouse {
      givenName = 'Mary'
      initial = 'T'
      familyName = 'Smith'
```

```
    }
    children[0] {
       name {
          givenName = 'Ralph'
          initial = 'T'
          familyName = 'Smith'
       }
       dateOfBirth = '19571111'
    }
    children[1] {
       name {
          givenName = 'Susan'
          initial = 'B'
          familyName = 'Jones'
       }
       dateOfBirth = '19590717'
    }
}
```

Generated Makefile

The *-genmake* option causes a makefile to be generated. The makefile can be used to compile the generated source files to create the asn1api.dll library. User can use the *nmake* command in Visual Studio .NET command prompt to use this makefile.

When a makefile is generated, it is assumed that the ASN1C project exists within the ASN1C installation directory tree. The generation logic tries to determine the root directory of the installation by traversing upward from the project directory in an attempt to locate the *csharp* subdirectory which is assumed to be the installation root directory. If the project is located outside of the ASN1C hierarchy, the user can set the OSROOTDIR environment variable to point at the root directory.

If the root directory is located successfully, the generated build script will use that directory; however, if the compiler fails to find the installation root directory, it will use @ROOT_DIR@ instead and print an error message. Users will have to manually replace @ROOT_DIR@ with the actual compiler installation root directory.

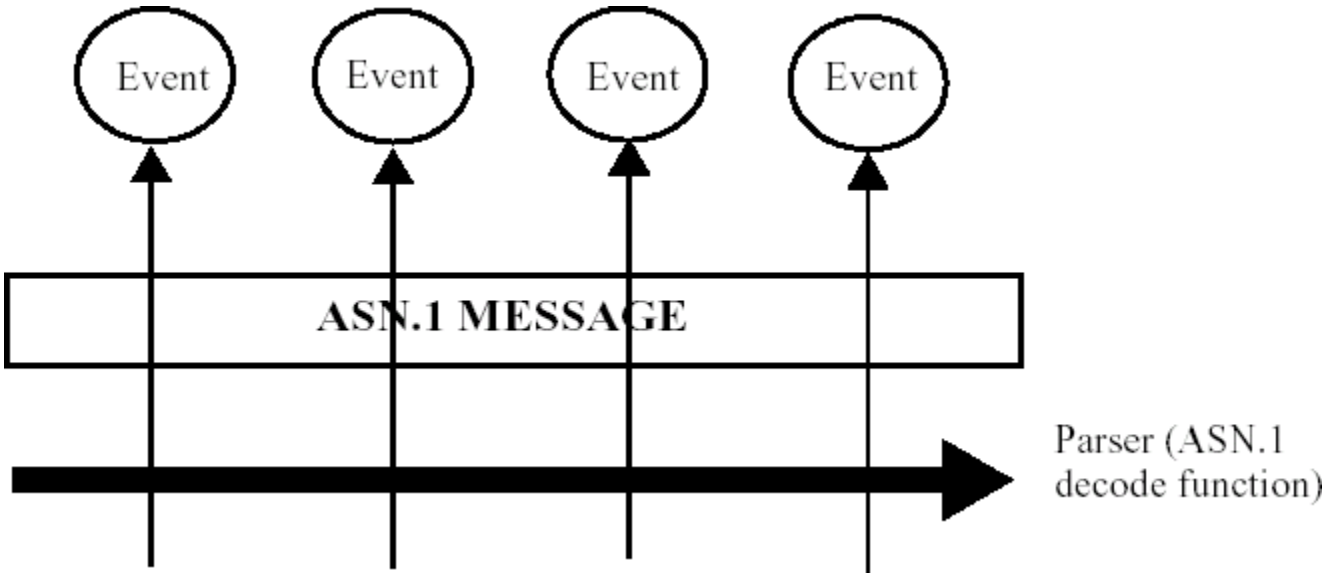
If the compiler fails to find the asn1rt.dll runtime library, it will use @ASN1RT_DIR@ instead and print an error message. User will have to manually replace @ASN1RT_DIR@ with the actual C# runtime library directory.

Event Handler Interface

The *–events* command line switch causes hooks for user-defined event handlers to be inserted into the generated C# decode methods. What these event handlers do is up to the user. They fire when key message-processing events occur during the course of parsing an ASN.1 message. They are similar in functionality to the Simple API for XML (SAX) that was described earlier for parsing XML messages.

How It Works

Users of XML parsers are probably already quite familiar with the concepts of SAX. Significant events are defined that occur during the parsing of a message. As a parser works through a message, these events are 'fired' as they occur by invoking user defined callback functions. These callback functions are also known as event handler functions. A diagram illustrating this parsing process is as follows:



The events are significant actions that occur during the parsing process. The following events are defined that will be passed to the user when an ASN.1 message is parsed:

1. **StartElement** – This event occurs when the parser moves into a new element. For example, if we have a SEQUENCE { a, b, c } construct (type names omitted), this event will fire when we begin parsing a, b, and c. The name of the element is passed to the event handling callback function.

2. **EndElement** – This event occurs when the parser leaves a given element space. Using the example above, these would occur after the parsing of a, b, and c are complete. The name of the element is once again passed to the event handling callback function.

3. **Characters method** – This method is defined to pass all of the different types of primitive values that are encountered when parsing a message. The primitive values are passed out in a stringified form.

The methods corresponding to these events are defined in *Asn1NamedEventHandler* interface.

The start and end element methods are invoked when an element is parsed within a constructed type. The start method is invoked as soon as the tag/length is parsed in a BER or DER message. The end method is invoked after the contents of the field are processed. The signature of these methods is as follows:

```
void StartElement (string name, int index);
```

```
void EndElement (string name, int index);
```

The *name* argument is used pass the element name. The *index* argument is used for SEQUENCE OF/SET OF constructs only. It is used to pass the index of the item in the array. This argument is set to –1 for all other constructs.

The *Characters* method is used to pass out ASN.1 primitive data. This is a departure from the C++ event handler methodology in which separate methods are defined for all of the different data types. This implementation is more closely aligned with the standard SAX implementation for XML. The reason it is done this way in C# and not C++ is because it is much easier to stringify values. Since memory management is built-in to C#, it is easy to create a string and pass it out. This is a problem in C++ because it becomes a performance issue if too many malloc's are done and it also places a burden on the user to free the memory for the allocated strings.

The signature for the *Characters* method is as follows:

```
void Characters (string svalue, short typeCode);
```

The *svalue* argument contains the stringified value. The format of this value is ASN.1 value notation for the value as defined in the X.680 standard. The *typeCode* argument contains an identifier that specifies the ASN.1 type of the value. The identifier corresponds to the universal identifier values (the ID number in the universal tags) for each of the primitive data types. The only exception to this rule is that the identifier 99 was added to represent an Open Type construct. Constants for all of the identifier values are provided in the *Asn1Type* class. See the C# documentation for this class for a list of the constants.

How to Use It

To define event handlers, two things must be done:

1. One or more new classes must implement the *Asn1NamedEventHandler* interface.

2. Objects of these classes must be created and registered prior to calling the generated decode method for a particular type.

The best way to illustrate this procedure is through examples. We will first show a simple event handler application to provide a customized formatted printout of the fields in a BER message. Then we will show a simple XML converter class that will convert the data in a BER message to XML.

Example 1: A Formatted Print Handler

The ASN1C evaluation and distribution kits include a sample program for doing a formatted print of parsed data. This code can be found in the *csharp/sample_ber/EventHandler directory*. Parts of the code will be reproduced here for reference, but refer to this directory to see the full implementation.

The format for the printout will be simple. Each element name will be printed followed by an equal sign (=) and an open brace ({) and newline. The value will then be printed followed by another newline. Finally, a closing brace (}) followed by another newline will terminate the printing of the element. An indentation count will be maintained to allow for a properly indented printout.

We will first create a class called *PrintHandler* that implements the *Asn1NamedEventHandler* interface and handles the formatted printing of the data. The rule for the implementation of interfaces is that you must provide an implementation for each of the methods listed. That is it. You can add as many additional methods, member variables, etc., that you like.

The *PrintHandler* implementation that we created is as follows:

```
class PrintHandler : Asn1NamedEventHandler {
   protected string mVarName;
   protected int mIndentSpaces = 0;
```

```
      public PrintHandler (string varName) {
         mVarName = varName;
         System.Console.Out.WriteLine (mVarName + " = {");
         mIndentSpaces += 3;
      }

      public void StartElement (string name, int index) {
         Indent();
         System.Console.Out.Write (name);
         if (index >= 0)
            System.Console.Out.Write ("[" + index + "]");
         System.Console.Out.WriteLine (" = {");
         mIndentSpaces += 3;
      }

      public void EndElement (string name, int index) {
         mIndentSpaces -= 3;
         Indent ();
         System.Console.Out.WriteLine ("}");
      }

      public void Characters (string svalue, short typeCode) {
         Indent ();
         System.Console.Out.WriteLine (svalue);
      }

      private void Indent () {
         for (int i = 0; i < mIndentSpaces; i++)
            System.Console.Out.Write (" ");
      }
   }
```

In this definition, we chose to add the mVarName and mIndentSpaces member variables to keep track of these items. The user is free to add any type of member variables he or she wants. The only firm requirement in defining this class is the implementation of the methods defined in the interface.

We implement these methods as follows:

In *StartElement*, we print the name, equal sign, and opening brace:

```
   public void StartElement (string name, int index) {
      Indent();
      System.Console.Out.Write (name);
      if (index >= 0)
         System.Console.Out.Write ("[" + index + "]");
      System.Console.Out.WriteLine (" = {");
      mIndentSpaces += 3;
   }
```

In this simplified implementation, we simply indent (this is another private method within the class) and print out the name, equal sign, and opening brace. We then increment the indent level. Logic is also present to check the index value to see if it is zero or greater. If it is, an array subscript is added to the element name.

In *EndElement*, we simply terminate our brace block as follows:

```
public void EndElement (string name, int index) {
   mIndentSpaces -= 3;
   Indent ();
   System.Console.Out.WriteLine ("}");
}
```

The *Characters* method simply indents and prints the stringified value:

```
public void Characters (string svalue, short typeCode) {
   Indent ();
   System.Console.Out.WriteLine (svalue);
}
```

That completes the *PrintHandler* class implementation.

Next, we need to create an object of the class and register it prior to invoking the decode method. In the *Reader.cs* program, the following lines do this:

```
// Register event handler object

PrintHandler printHandler = new PrintHandler ("personnelRecord");
decodeBuffer.AddNamedEventHandler (printHandler);
```

The *addEventHandler* method defined in the *Asn1DecodeBuffer* base class is the mechanism used to do this. Note that event handler objects can be stacked. Several can be registered before invoking the decode function. When this is done, the entire list of event handler objects is iterated through and the appropriate event handling callback function invoked whenever a defined event is encountered.

The implementation is now complete. The program can now be compiled and run. When this is done, the resulting output is as follows:

```
employee = {
   name = {
      givenName = {
         "John"
      }
      initial = {
         "P"
      }
      familyName = {
         "Smith"
      }
   }
   ...
```

This can certainly be improved. For one thing it can be changed to print primitive values out in a “name = value” format (i.e., without the braces). But this should provide the general idea of how it is done.

Example 2: An XML Converter Class

The ASN1C XML Encoding Rules (XER) encode and decode capabilities were presented in an earlier section of this document. An alternate way to create an XML document from ASN.1 data is through the event handler interface.

It turns out that with event handlers, this conversion is fairly easy. As the handler events fire, all of the required symbolic data is passed out to generate an XML document. The programmer is free to massage this data any way he or she wants to comply with whatever DTD or XML Schema is in use.

The *ToXML* sample program demonstrates the conversion of ASN.1 data to XML using event handlers. The sample is not intended to be a robust implementation – it is merely designed to provide guidance in how one would go about doing this transformation.

The sample program can be found in the *csharp/sample_ber/ToXML* subdirectory within the ASN1C installation. The complete class definition for the *XMLHandler* class is as follows:

```
class XMLHandler : Asn1NamedEventHandler {
   protected string mVarName;
   protected int mIndentSpaces = 0;

   public XMLHandler (string varName) {
      mVarName = varName;
      System.Console.Out.WriteLine ("<" + mVarName + ">");
      mIndentSpaces += 3;
   }

   public void StartElement (string name, int index) {
      Indent();
      System.Console.Out.WriteLine ("<" + name + ">");
      mIndentSpaces += 3;
   }

   public void EndElement (string name, int index) {
      mIndentSpaces -= 3;
      Indent ();
      System.Console.Out.WriteLine ("</" + name + ">");
   }

   public void Characters (string svalue, short typeCode) {
      Indent ();
      string typeName = new string (Asn1Type.getTypeName(typeCode));
      typeName.Replace (' ', '_');
      System.Console.Out.Write ("<" + typeName + ">");
      System.Console.Out.Write (svalue);
      System.Console.Out.WriteLine ("</" + typeName + ">");
   }

   public void Finished () {
      System.Console.Out.WriteLine ("</" + mVarName + ">");
   }

   private void Indent () {
      for (int i = 0; i < mIndentSpaces; i++)
      System.Console.Out.Write (" ");
   }
}
```

This is very similar to the *PrintHandler* class defined earlier. The *StartElement* method simply opens an XML element block:

```
public void StartElement (string name, int index) {
   Indent();
   System.Console.Out.WriteLine ("<" + name + ">");
   mIndentSpaces += 3;
```

```
}
```

The *EndElement* method closes it:

```
public void EndElement (string name, int index) {
   mIndentSpaces -= 3;
   Indent ();
   System.Console.Out.WriteLine ("</" + name + ">");
}
```

The *Characters* method outputs the data with a type wrapper:

```
public void Characters (string svalue, short typeCode) {
   Indent ();
   string typeName = new string (Asn1Type.getTypeName(typeCode));
   typeName.Replace (' ', '_');
   System.Console.Out.Write ("<:" + typeName + ">");
   System.Console.Out.Write (svalue);
   System.Console.Out.WriteLine ("</:" + typeName + ">");
}
```

This illustrates the use of the *typeCode* argument for obtaining information on the ASN.1 type of the data. Note that this is a simplified version of an XER formatting method. A true implementation would need to do some massaging of the stringified data to fit the XER rules which, in general, do not follow the ASN.1 value formatting rules. The implementation would also need some logic to check if the type wrapper should be output or not; it is not always done in certain cases.

Finally note the constructor and *finished* method. The constructor prints out the outer-level wrapper tag. Since C# does not have destructors, a finished method is defined to terminate this tag. This method must be called manually from within the application program after the C# decode method. See the *Reader.cs* program to see how this is done.

Object registration is done as before in the *PrintHandler* example. The only difference is that an object of the *XML-Handler* class is created instead of the *PrintHandler* class.

When compiled and executed, the output from the Reader program looks like this:

```
<PersonnelRecord>
   <name>
      <givenName>
         <IA5String>'John'</IA5String>
      </givenName>
      <initial>
         <IA5String>'P'</IA5String>
      </initial>
      <familyName>
         <IA5String>'Smith'</IA5String>
      </familyName>
   </name>
   <number>
      <INTEGER>51</INTEGER>
   </number>
   <title>
      <IA5String>'Director'</IA5String>
   </title>
   <dateOfHire>
```

```
         <IA5String>'19710917'</IA5String>
      </dateOfHire>
      <nameOfSpouse>
         <givenName>
            <IA5String>'Mary'</IA5String>
         </givenName>
         <initial>
            <IA5String>'T'</IA5String>
         </initial>
         <familyName>
            <IA5String>'Smith'</IA5String>
         </familyName>
      </nameOfSpouse>
      <children>
         <element>
            <name>
               <givenName>
                  <IA5String>'Ralph'</IA5String>
               </givenName>
               <initial>
                  <IA5String>'T'</IA5String>
               </initial>
               <familyName>
                  <IA5String>'Smith'</IA5String>
               </familyName>
            </name>
            <dateOfBirth>
               <IA5String>'19571111'</IA5String>
            </dateOfBirth>
         </element>
         <element>
            <name>
               <givenName>
                  <IA5String>'Susan'</IA5String>
               </givenName>
               <initial>
                  <IA5String>'B'</IA5String>
               </initial>
               <familyName>
                  <IA5String>'Jones'</IA5String>
               </familyName>
            </name>
            <dateOfBirth>
               <IA5String>'19590717'</IA5String>
            </dateOfBirth>
         </element>
      </children>
   </PersonnelRecord>
```

Add an XML document header and you should be able to display this data in XML-enabled browser.

IMPORT/EXPORT of Types

ASN1C allows productions to be shared between different modules through the ASN.1 IMPORT/EXPORT mechanism. The compiler parses but ignores the EXPORTS declaration within a module. As far as it is concerned, any type defined within a module is available for import by another module.

When ASN1C sees an IMPORT statement, it first checks its list of loaded modules to see if the module has already been loaded into memory. If not, it will attempt to find and parse another source file containing the module. The logic for locating the source file is as follows:

1. The configuration file (if specified) is checked for a <sourceFile> element containing the name of the source file for the module.

2. If this element is not present, the compiler looks for a file with the name <ModuleName>.asn where module name is the name of the module specified in the IMPORT statement.

In both cases, the –I command line option can be used to tell the compiler where to look for the files.

The other way of specifying multiple modules is to include them all within a single ASN.1 source file. It is possible to have an ASN.1 source file containing multiple module definitions in which modules IMPORT definitions from other modules. An example of this would be the following:

```
ModuleA DEFINITIONS ::= BEGIN
   IMPORTS B From ModuleB;

   A ::= B


END

ModuleB DEFINITIONS ::= BEGIN

   B ::= INTEGER
END
```

This entire fragment of code would be present in a single ASN.1 source file.

Compact Code Generation

The -*compact* command line switch can be used to reduce the amount of source code generated for a given ASN.1 specification. This is done by generating the code for simple definitions inline within structured type definitions instead of creating separate classes.

For example, consider the following definition:

```
X ::= [APPLICATION 1] INTEGER

Y ::= [APPLICATION 2] OCTET STRING (SIZE (1..32))

Z ::= [APPLICATION 3] SEQUENCE {
      x     [0] X,
      y     [1] Y
}
```

In normal mode, the compiler would generate three classes for these productions: one corresponding to X, Y, and Z respectively. But in compact mode, it is recognized that a user would normally not be interested in encoding or decoding X and Y on their own. They would primarily be interested in encoding or decoding the more complex structured types (i.e. the PDU's) that make up fully formed messages. Taking this into account, when *–compact* is specified, the compiler will not generate separate classes for X and Y in the above definition. Instead, it will include only the base types for X and Y in the generated code for the SEQUENCE Z. All logic to handle the tags and constraints will be built directly into the Z encode and decode methods.

So the result will be only a single class generated (Z) that will contain an Asn1Integer object to represent X and an Asn1OctetString object to represent Y. The logic to process the application tags and the size constraint on the octet string will be generated inline in the encode and decode methods in Z.

Due to the way some ASN.1 specifications are written, this can have a significant effect in reducing the amount of generated code. For example, in the TAP3 sample program, the total number of generated classes was reduced from 20 to 3.

ASN1C90

As of ASN1C version 5.7, ASN1C90 is now an option built-in to the standard ASN1C executable program. This option is **-1990**. Therefore, specifying ‘asn1c <file> -1990’ on the command-line is equivalent to using the old ‘asn1c90’ executable program. The remaining text in this section refers to using ASN1C with the -1990 option.

ASN1C when used with the -1990 option contains extensions to handle the older 1990 version of ASN.1. Although this version is no longer supported by the ITU-T, it is still in use today.

This version of the compiler also contains logic to parse some common MACRO definitions that are still in widespread use despite the fact that MACRO syntax was retired with this version of the standard. The types of MACRO definitions that are supported are ROSE OPERATION and ERROR and SNMP OBJECT-TYPE.

ROSE OPERATION and ERROR

ROSE stands for “Remote Operations Service Element” and defines a request/response transaction protocol in which requests to a conforming entity must be answered with the result or errors defined in operation definitions Variations of this are used in a number of protocols in use today including CSTA and TCAP.

The definition of the ROSE OPERATION MACRO that is built into the ASN1C90 version of the compiler is as follows:

```
OPERATION MACRO ::=
BEGIN
   TYPE NOTATION          ::= Parameter Result Errors LinkedOperations
   VALUE NOTATION         ::= value (VALUE INTEGER)
   Parameter              ::= ArgKeyword NamedType | empty
   ArgKeyword             ::= "ARGUMENT" | "PARAMETER"
   Result                 ::= "RESULT" ResultType | empty
   Errors                 ::= "ERRORS" "{"ErrorNames"}" | empty
   LinkedOperations       ::= "LINKED" "{"LinkedOperationNames"}" | empty
   ResultType             ::= NamedType | empty
   ErrorNames             ::= ErrorList | empty
   ErrorList              ::= Error | ErrorList "," Error
   Error                  ::= value(ERROR)        -- shall reference an error value
                              | type              -- shall reference an error type
                                                  -- if no error value is specified
   LinkedOperationNames   ::= OperationList | empty
   OperationList          ::= Operation | OperationList "," Operation
   Operation              ::= value(OPERATION)    -- shall reference an operation va
                              | type              -- shall reference an operation ty
                                                  -- if no operation value is specif
   NamedType              ::= identifier type | type

END
```

This MACRO does not need to be defined in the ASN.1 specification to be parsed. In fact, any attempt to redefine this MACRO will be ignored. Its definition is hard-coded into the compiler.

What the compiler does with this definition is uses it to parse types and values out of OPERATION definitions. An example of an OPERATION definition is as follows:

```
login OPERATION
ARGUMENT SEQUENCE { username IA5String, password IA5String }
```

```
RESULT SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String }
ERRORS { authenticationFailure, insufficientResources }
::= 1
```

In this case, there are two embedded types (an ARGUMENT type and a RESULT type) and an integer value (1) that identifies the OPERATION. There are also error definitions.

The ASN1C90 compiler generates two types of items for the OPERATION:

1. It extracts the type definitions from within the OPERATION definitions and generates equivalent C# classes and encoders/decoders, and

2. It generates value constants for the value associated with the OPERATION (i.e., the value to the right of the '::=' in the definition).

The compiler does not generate any structures or code related to the OPERATION itself (for example, code to encode the body and header in a single step). The reason is because of the multi-layered nature of the protocol. It is assumed that the user of such a protocol would be most interested in doing the processing in multiple stages, hence no single function or structure is generated.

Therefore, to encode the login example the user would do the following:Therefore, to encode the login example the user would do the following:

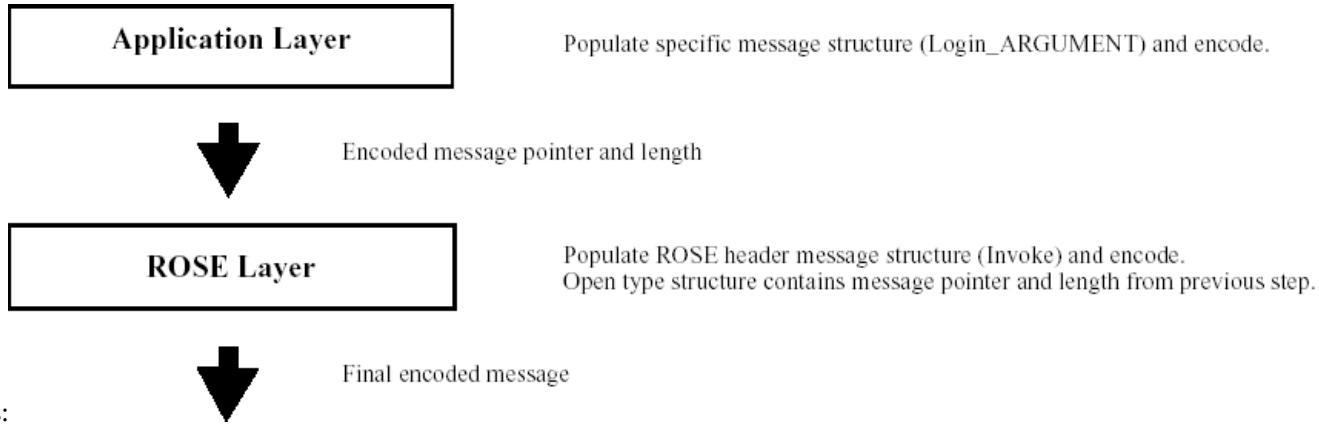
1. At the application layer, the Login_ARGUMENT structure would be populated with the username and password to be encoded.

2. The encode function for Login_ARGUMENT would be called and the resulting message pointer and length would be passed down to the next layer (the ROSE layer).

3. At the ROSE layer, the Invoke structure would be populated with the OPERATION value, invoke identifier, and other header parameters. The open type object used to hold the encoded parameter value from step 2 is populated by creating an Asn1OpenType object using the length of the encoded component.

4. The encode function for Invoke would be called resulting in a fully encoded ROSE Invoke message ready for transfer across the communications link.

The following is a picture showing this process:


Application Layer
Populate specific message structure (Login_ARGUMENT) and encode.
Encoded message pointer and length
ROSE Layer
Populate ROSE header message structure (Invoke) and encode.
Open type structure contains message pointer and length from previous step.
Final encoded message

On the decode side, the process would be reversed with the message flowing up the stack:

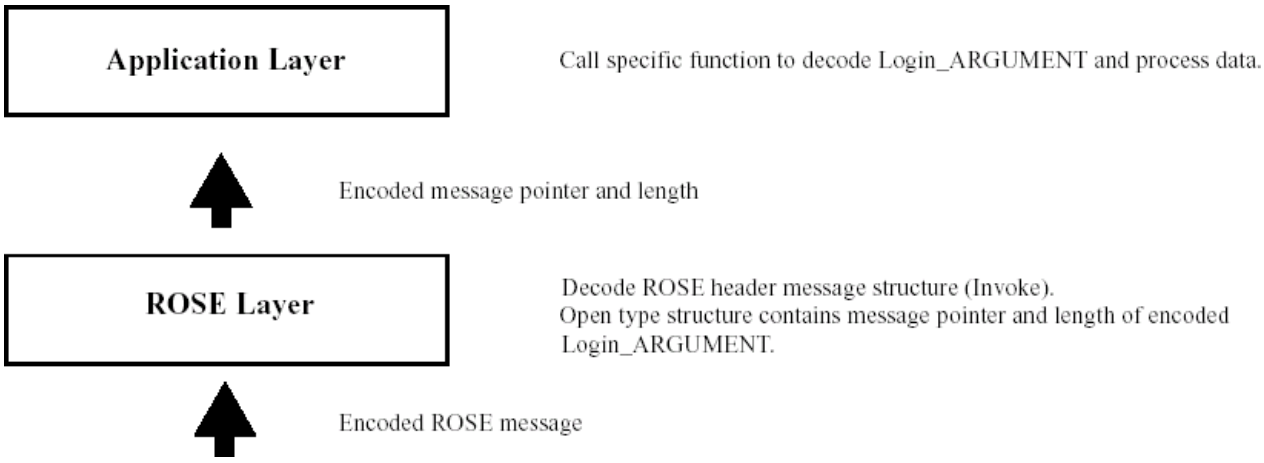
1. At the ROSE layer, the header would be decoded producing information on the OPERATION type (based on the MACRO definition) and message type (Invoke, Result, etc..). The invoke identifier would also be available for use in session management. In our example, we would know at this point that we got a login invoke request.

2. Based on the information from step 1, the ROSE layer would know that the Open Type field contains a pointer and length to an encoded Login_ARGUMENT component. It would then route this information to the appropriate processor within the Application Layer for handling this type of message.

3. The Application Layer would call the specific decoder associated with the Login_ARGUMENT. It would then have available to it the username/password the user is logging in with. It could then do whatever application-specific processing is required with this information (database lookup, etc.).

4. Finally, the Application Layer would begin the encoding process again in order to send back a Result or Error message to the Login Request.

A picture showing this is as follows:


Application Layer
Call specific function to decode Login_ARGUMENT and process data.
Encoded message pointer and length
ROSE Layer
Decode ROSE header message structure (Invoke).
Open type structure contains message pointer and length of encoded Login_ARGUMENT.
Encoded ROSE message

The login OPERATION also contains references to ERROR definitions. These are defined using a separate MACRO that is built into the compiler. The definition of this MACRO is as follows:

```
ERROR MACRO ::=
BEGIN
   TYPE NOTATION ::= Parameter

   VALUE NOTATION ::= value (VALUE INTEGER)

   Parameter ::= "PARAMETER" NamedType | empty

   NamedType ::= identifier type | type

END
```

In this definition, an error is assigned an identifying number as well as on optional parameter type to hold parameters associated with the error. An example of a reference to this MACRO for the `authenticationFailure` error in the login operation defined earlier would be as follows:

```
applicationError ERROR
PARAMETER SEQUENCE {
   errorText IA5String
}
::= 1
```

The ASN1C90 compiler will generate a type definition for the error parameter and a value constant for the error value. The format of the name of the type generated will be “<name>_PARAMETER” where <name> is the ERROR name

(applicationError in this case) with the first letter set to uppercase. The name of the value will simply be the ERROR name.

SNMP OBJECT-TYPE

The SNMP OBJECT-TYPE MACRO is one of several MACROs used in Management Information Base (MIB) definitions. It is the only MACRO of interest to ASN1C because it is the one that specifies the object identifiers and data that are contained in the MIB.

The version of the MACRO currently supported by this version of ASN1C can be found in the SMI Version 2 RFC (RFC 2578). The compiler generates code for two of the items specified in this MACRO definition:

1. The ASN.1 type that is specified using the SYNTAX command, and

2. The assigned OBJECT IDENTIFIER value

For an example of the generated code, we can look at the following definition from the UDP MIB:

```
udpInDatagrams OBJECT-TYPE
   SYNTAX      Counter32
   MAX-ACCESS  read-only
   STATUS      current
   DESCRIPTION
      "The total number of UDP datagrams delivered to UDP users."
   ::= { udp 1 }
```

In this case, a type definition is generated for the SYNTAX element and an Object Identifier value is generated for the entire item. The name used for the type definition is “<name>_SYNTAX” where <name> would be replaced with the OBJECT-TYPE name (i.e., udpInDatagrams). The name used for the Object Identifier value constant is the OBJECTTYPE name. So for the above definitions, the following two C# items would be generated:

1. A “udpInDatagrams_SYNTAX.cs” file. This would contain the udpInDatagrams_SYNTAX class definition, and

2. A udpInDatagrams value definition in the _UDP_MIBValues class.